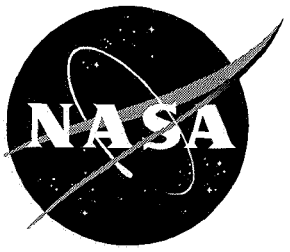


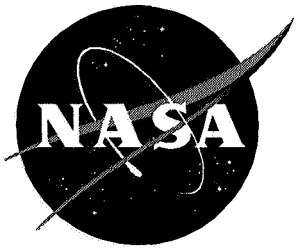
NASA Contractor Report 4594

Interpreter Composition Issues in the Formal Verification of a Processor-Memory Module

David A. Fura and Gerald C. Cohen

Contract NAS1-18586
Prepared for Langley Research Center

May 1994



NASA Contractor Report 4594

Interpreter Composition Issues in the Formal Verification of a Processor-Memory Module

David A. Fura and Gerald C. Cohen
The Boeing Company • Seattle, Washington

National Aeronautics and Space Administration
Langley Research Center • Hampton, Virginia 23681-0001

Prepared for Langley Research Center
under Contract NAS1-18586

May 1994

Preface

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems Suitable for Fly-By-Wire Applications, Task Assignment 12. Task 12 is concerned with issues in the formal specification and verification of a processor-memory module.

This report describes interpreter composition techniques suitable for the formal specification and verification of a processor-memory module using the HOL theorem-proving system. The processor-memory module is a multi-chip subsystem within a fault-tolerant embedded system under development within the Boeing Defense & Space Group. It provides the opportunity to investigate the specification and verification of a real-world subsystem within a commercially-developed fault-tolerant computer.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was accomplished at the Boeing Company, Seattle, Washington. Personnel responsible for the work include:

Boeing Defense & Space Group:
D. Gangsaas, Responsible Manager
T. M. Richardson, Program Manager

Boeing Defense & Space Group:
Gerald C. Cohen, Principal Investigator
David A. Fura, Researcher

Contents

List of Figures

List of Tables

1 Introduction

To date, theorem-proving efforts targeting microprocessor-based system verification have treated these systems as self-enclosed state-transition systems without outputs (e.g., [Hun86], [Coh88], [Joy90], [Win90], [Gra92], [Lev93]). While these approaches have been effective on the small-scale systems addressed thus far, they are not ones that we expect to scale up to larger systems in any practical way. The fundamental problem with these approaches is their insistence on performing subsystem compositions at very low levels of abstraction.

We believe that the ability to compose hardware models at high levels of abstraction is fundamental to the practical specification and formal verification of nontrivial hardware systems. Without such a capability, large-scale system verifications would quickly become bogged down by the explosion in low-level state resulting from the independent actions of typical interacting subsystems. In this report we describe a new approach to interpreter composition that permits provably-secure compositions at high levels of abstraction.

Our work under this task is ultimately targeted to a commercially-developed processing subsystem, called the Processor-Memory Module (or PMM), of a fault-tolerant computer system. The Fault Tolerant Embedded Processor (FTEP) is targeted towards applications in avionics and space requiring extremely high levels of mission reliability, extended maintenance-free operation, or both. Since the need for high-quality design assurance in these systems is an undisputed fact, the continued development and application of formal methods is vital as these systems see increasing use in modern society.

The work described in this report represents the culmination of a multiyear effort to develop and apply theorem-proving technology to hardware verification. Previous work under Task 10 developed abstraction techniques and an interpreter modeling approach to specify a PMM interface chip (the Processor Interface Unit, or PIU) in terms of its higher-level transaction handling. In this context, a transaction refers to the bus transactions of typical commercial microprocessors, such as the Intel 80960 [Int89] or the MIPS R3000 [Kan87]. Techniques for formally verifying the transaction level with respect to its underlying clock level were also developed in Task 10. Two Task 10 final reports describe the results of the PIU specification and verification work ([Fur93a] and [Fur93b], respectively).

Prior to this, work under Task 3 developed interpreter modeling and verification techniques applicable to a wide range of hardware verification problems. A new approach to hierarchical decomposition and the development of a generic interpreter theory together greatly reduced the level of effort required by many verifications. The recent verification of an implementation of the Viper microprocessor certifies the strength of these methods. The methodology that was employed in this verification is described in [Win91] and [Win92], while the verification itself is described in [Lev93].

The research focus of the Task 12 work described here was on composition, specifically *transaction-level composition.* We have developed modeling and verification methods that permit provably-secure composition at this high level of abstraction. To our knowledge, this work represents the first successful demonstration of an interpreter-based approach to this problem. All of our work was performed using the HOL theorem proving system from the University of Cambridge [Gor93].

Section 2 of this report describes an interpreter modeling approach first developed under Task 10. Called 'hierarchical pre-post logic' (or HPL), this model has seen further refinement during Task 12. The aspects of HPL covered here include the modeling of interpreter behavior, interpreter abstraction, interpreter liveness, and interpreter correctness.

Section 3 explains important issues in secure abstract-level composition. The primary focus is on the role that abstraction plays in secure abstract-level composition. Low-level hardware composition is reviewed and a new approach to abstract-level synchronization, based on interface-event clocking, is introduced.

Section 4 describes how interface-event clocking impacts the modeling and verification of interpreter-based systems. It introduces 'multirate' interpreters, where the interpreter inputs and outputs clock at different rates. Largely through two significant examples, techniques to solve some important modeling and verification problems are demonstrated.

Section 5 contains a description of the new PMM specification models produced under this task and describes some modifications to the existing models required to support composition.

Section 6 presents our conclusions.

A brief description of the HOL theorem proving system is contained in Appendix A.

Appendix B contains the HOL listings of several interpreter verification test case studies.

Appendix C provides Processor-Memory Module specification examples.

2 Hierarchical Pre-Post Logic

In this section we describe a formal modeling approach that addresses the specification and verification needs of the PMM. The interpreter behavioral model provides for the specification of both operation preconditions and postconditions, hence the 'pre-post' designation. The approach supports hierarchical decomposition of system specifications and the explicit relating of the different levels through the use of abstraction predicates, as explained in this section.

In describing this hierarchical pre-post logic (HPL), we make use of a specification approach introduced by Joyce [Joy90], by describing operators and data types *generically*. Many operators are defined only by their types, with nothing said about what operations they actually perform. In this way, theorems proven about them are applicable to all operators satisfying the appropriate type. We make extensive use of type variables so that our results are applicable, in some instances, to structures with arbitrary types.

In this section we make consistent use of the following variable names. The variables **k**, **t**, **s**, **e**, and **p** represent, in order, instructions, time, state signals, environment (input) signals, and output signals. When a variable is primed it denotes a *concrete*-level, or implementing, entity. Type variables prefixed by a * are polymorphic and denote abstract types. Note that each of the types shown here is abstract except for that of the time variable. The type **":time"** is an abbreviation for the HOL type for natural numbers **":num"**

Common Variables and Their Types

Instructions:	**k :*instr, k' :*instr'**
Time:	**t :time, t' :time'**
State:	**s :time→*state, s' :time'→*state'**
Environment:	**e :time→*env, e' :time'→*env'**
Output:	**p :time→*out, p' :time'→*out'**

The following generic operators are used throughout this section. We don't describe them in detail here, but merely point out that the first three (the execution predicate, precondition, and postcondition) are associated with the interpreter behavioral model (Section 2.1) and the last four (event predicate, state, input, and output abstraction) are used in the abstraction definition (Section 2.2).

The variable **rep** is an n-tuple whose elements are of the types specified for the generic operators. The operators are implemented as accessor functions that select elements of this n-tuple. For example, **EXEC** is a function that, when applied to **rep**, returns an arbitrary value of the appropriate type.

Generic Operators

Execution Predicate:	**EXEC rep**	**:*instr→s_sig→e_sig→p_sig→time→bool**
Precondition:	**PREC rep**	**:*instr→s_sig→e_sig→p_sig→time→bool**
Postcondition:	**POSTC rep**	**:*instr→s_sig→e_sig→p_sig→time→bool**
Event Predicate:	**G rep**	**:time'→bool**
State Abstraction:	**SAbs rep**	**:s_sig'→e_sig'→p_sig'→time'→*state**
Environment Abstraction:	**EAbs rep**	**:s_sig'→e_sig'→p_sig'→time'→*env**
Output Abstraction:	**PAbs rep**	**:s_sig'→e_sig'→p_sig'→time'→*out**

where:

s_sig = (time→*state)	**s_sig' = (time'→*state')**
e_sig = (time→*env)	**e_sig' = (time'→*env')**
p_sig = (time→*out)	**p_sig' = (time'→*out')**

The rest of this section describes four major aspects of our modeling approach: (a) the interpreter behavior model, (b) abstraction between levels, (c) interpreter liveness, and (d) interpreter correctness. Each of these is covered in its own subsection.

2.1 Interpreter Behavioral Specification

In this and the subsequent sections we begin first with a description of prior work, usually with the HOL system. After this we describe the HPL approach to the particular subject being addressed.

2.1.1 Related Work

Interpreter modeling has a significant history now within the theorem proving community. In this section we discuss three different approaches that have been implemented within the HOL system, as well as others. Because the term 'interpreter' is such a broad one, covering essentially the entire class of finite-state machines, we have limited our focus to those approaches having been used to model hardware at higher levels of abstraction. By 'higher' we mean above the level of latches and flip-flops. The approaches cited here have all been used for modeling microprocessor instruction sets, for example. We have made minor modifications to the original definitions in some cases to achieve a common syntax for this section.

'FSM' Approach

The first approach was used by a number of researchers: Cohn for the Viper proof [Coh88][Coh89], Joyce for the Tamarack proof [Joy90], and Graham for the SECD proof [Gra92]. In each of these microprocessor modeling and verification efforts, the machine behavior was defined using a single function to represent the next state with respect to the current state and current environment. Interpreter outputs were not modeled.

FSM Specification Style [Coh88][Joy90][Gra92]

```
∀ t . s (t+1) = NEXT_STATE (s t) (e t)
```

This approach is a natural one to take and, indeed, worked as intended for all three of the cited efforts. However, one difficulty with it is its embedding of the instruction decoding operation within the next-state function. As explained in Section 2.4, this leads to some complexity in performing an implementation correctness proof.

GIT Approach

Windley developed the generic interpreter theory (GIT) to address some of the problems existing with the previous interpreter models [Win90]. The basic idea behind the theory is that by implementing (inside the theory) some of the difficult proofs common to many interpreter verifications, the user is spared the burden of carrying out these proofs. Furthermore, the theory presents to the user a well-defined interface, detailing exactly what kinds of specifications need to be made and precisely what lemmas need to be proved. This has the effect of greatly streamlining the specification and verification tasks associated with interpreter proofs that use the theory.

The GIT specification style, shown next, is different from the FSM style in its use of an instruction variable, **k**, which is defined by the function **SELECT**. **SELECT** takes the current state and environment and returns an instruction, or 'instruction key,' in the GIT parlance. **SELECT** can be thought of as an instruction *decoding* function.

The instruction key is used by the **INSTRUCTIONS** and **OUTPUT** functions to determine the particular next-state and output functions, respectively, to use for the current time. These functions are denoted **NEXT_-**

STATE_K and **OUTPUT_K** here to indicate that they are associated with a single instruction, **k**, and do not define the behavior for the entire instruction set.

GIT Specification Style [Win90]

```
∀ t . let k = SELECT (s t) (e t) in
      let NEXT_STATE_K = INSTRUCTIONS k in
      let OUTPUT_K = OUTPUT k in
      (s (t+1) = NEXT_STATE_K (s t) (e t) ∧
       p t = OUTPUT_K (s t) (e t))
```

From a specification standpoint, the GIT approach is very similar to the previous FSM models in its reliance on functions to define interpreter behavior. The incorporation of outputs into the GIT is a natural step forward however. Section 2.4 explains in more detail the advantages of the GIT approach over the previous models in its support for interpreter verification. /

RTS Approach

In [Her92], Herbert describes a relational transition system (RTS), a modified GIT in which the behavior is expressed with relations rather than functions. One advantage of such a modification is that it admits *partial* descriptions of behavior. This was an important consideration in Herbert's work on incremental design verification, where proofs were sometimes being performed before the design was finished.

The RTS specification style is shown in the following HOL code. To begin, the approach makes use of a property list, **prop_list**, where each property, **prop**, within this list is a 2-tuple. The first element of the 2-tuple is an instruction key and the second element is a predicate. The main idea is that, for each execution step, exactly one key will be selected and the predicate it is paired with acts as a 'postcondition' for the execution step. The predicate **IS_SELECT (k, s t, e t)** is a relational version of the GIT **SELECT** function. It returns true when the instruction key, **k**, is selected by the current state, **s t**, and environment, **e t**. The postcondition predicate specifies the correct relationship between the next state and the current state and environment.

Interpreter behavior is expressed as: for every property, **prop**, if **prop** is a member of the property list, and if the key within **prop** matches that selected by **IS_SELECT**, then the postcondition within **prop** is satisfied.

RTS Specification Style [Her92]

```
∀ t. let k = ε k. IS_SELECT (k, s t, e t) in
     (∀ prop. prop MEM prop_list ∧
              (FST prop = k)
              ⊃
              SND prop (s (t+1), s t, e t)))
```

where: **prop_list** *is a list of (instruction tag, predicate) pairs with type:*
":(*instr # (*state#*state#*env→bool)) list"

The use of the choice operator (ε) here is to guard against the possibility that multiple instruction keys are selected at a single time. Such a possibility would constitute a specification error and would be caught by this approach since the 'uniqueness' required to process the choice operator would not be established.

2.1.2 HPL Interpreter Behavior

Our interpreter behavioral model has benefitted from the prior work done on the FSM and GIT approaches. Herbert's efforts proceeded in parallel with ours, however, and thus our modeling work was performed independently of it.

As was the case for Herbert, the motivation for developing a new interpreter model here was certain limitations within the existing approaches. In the case of HPL, the objective was to model fault-tolerant systems at very high levels of abstraction. The model required three elements: (1) a predicate to represent the state of a system prior to a mission, (2) a predicate to represent a set of fault-occurrence properties, and (3) a predicate to represent the system state at mission end. The existing models did not provide this.

This work on fault-tolerance modeling was performed outside of this contract, and is described in detail in [Fur94]. It produced the basic structure of the model, with the three elements mentioned above given the names *precondition*, *execution predicate*, and *postcondition*, respectively. However, most of the work described in this section, and virtually all of that described in the following sections, was performed under Tasks 10 and 12 of this contract.

Interpreter Definition

To specify hardware interpreters, HPL employs only two of the three predicates directly: the execution predicate (**EXEC rep**) and the postcondition (**POSTC rep**) as shown in the following definition. Interpreter correctness is stated informally as "whenever an instruction is executed its postcondition is satisfied." This represents the standard behavior used in the other models described above. Where HPL differs is the combination of: (a) the use of an explicit instruction variable, **k**, (b) the separation of instruction decoding and execution into their own predicates, and (c) the relational-style postcondition predicate.

As explained in Section 2.4, providing the instruction variable **k** as part of the interpreter definition permits straightforward instruction set case splitting, thus simplifying correctness proofs. This is achieved without having to perform nontrivial theorem proving within a generic theory.

As explained later in this section, and in the interpreter liveness discussion in Section 2.3, having access to the instruction decoding behavior (the execution predicate) provides certain modeling advantages that can be exploited.

Because the postcondition has very little predefined structure, HPL provides a great amount of flexibility in its interpreter modeling. It is equivalent to RTS in this regard. It can accommodate the functional style of the FSM and GIT approaches, as well as the relational style of the RTS approach.

Intepreter Definition:

$$\vdash \mathbf{INTRP\ rep\ s\ e\ p} = \forall\, \mathbf{k\ t}.\ \mathbf{EXEC\ rep\ k\ s\ e\ p\ t} \supset \mathbf{POSTC\ rep\ s\ e\ p\ t}$$

Preconditioned Interpreter Definition

The above definition faithfully models the interpreter behavior we are interested in, and for many verifications it is suitable for direct use. There are some verification problems, however, that are inconvenient to solve using this model directly. In these proofs, correct execution requires that certain state variables contain specified initial values at the beginning of an operation. The transaction-level proofs for the PMM are

one example of this. A convenient way to handle these proofs is to first postulate the appropriate values using a precondition (**PREC rep**) and then prove the simpler theorem shown next.

Preconditioned Interpreter Definition:

```
⊢ INTRP_PREC rep s e p =
    ∀ k t . EXEC rep k s e p t ∧ PREC rep k s e p t ⊃ POSTC rep k s e p t
```

The need for operation preconditioning results in an additional theory obligation. The precondition must be initially established at time **0**, and then it must be propagated at each successor time. The following definition of 'precondition satisfaction' describes this property.

Precondition Satisfaction:

```
⊢ PREC_SAT rep s e p =
  (∀ k . EXEC rep k s e p 0 ⊃ PREC rep k s e p 0) ∧
  (∀ k k1 t . POSTC rep k s e p t ∧ EXEC rep k1 s e p (SUC t) ⊃ PREC rep k1 s e p (SUC t))
```

PREC_SAT is a property that clearly must be met if the precondition is to hold at every operation boundary. The reason for including the execution predicate here is that, in general, a state's value may be defined conditionally on an input occurrence that is encapsulated within the execution predicate.

While the need for the above condition is expected in a correctness proof, the following proof obligation did surprise us initially. It states that when an instruction is executed at some (nonzero) time then there must have been an instruction executed at the previous time.

Instruction Sequence Liveness:

```
⊢ SEQ_LIVE rep s e p = ∀ k t . EXEC rep k s e p (SUC t) ⊃ (∃ k1 . EXEC rep k1 s e p t)
```

'Instruction sequence liveness' is an issue for us because we are verifying an instruction set rather than a program. In a program we know that a prior instruction is executed by virtue of its position within the code. Instruction set verification does not permit this solution, instead we must explicitly prove it.

In the approach described here, interpreter correctness can be verified by satisfying the following three proof obligations. The justification theorem states that when they are satisfied, then the interpreter is correct The actual HOL definitions and the proof for this justification theorem are contained in Appendix B.

Modified Proof Obligations:

```
∀ rep s e p . INTRP_PREC rep s e p
∀ rep s e p . PREC_SAT rep s e p
∀ rep s e p . SEQ_LIVE rep s e p
```

Justification Theorem:

```
⊢ ∀ rep s e p .
    INTRP_PREC rep s e p ⊃
     PREC_SAT rep s e p ⊃
      SEQ_LIVE rep s e p ⊃
       INTRP rep s e p
```

In the tradition of Windley's generic interpreter theory, we believe that the breakdown of theorem proving tasks into two distinct classes, as shown here, has a great practical benefit in reducing the number of theorem proving tasks that must be repeated in every interpreter verification. The advantage to proving interpreter correctness this way is that the justification-theorem proof could be embedded within a generic theory and then simply reused whenever a new interpreter is being verified. The user would only be concerned with proving facts that were specific to the particular circuit being addressed. In this case, these are the proof obligations shown here.

2.2 Interpreter Abstraction

Virtually all interpreter verifications performed in HOL have used the abstraction ideas described by Melham in [Mel90]. Of the different types of abstraction described in this previous work, two are of primary interest to interpreter modeling and verification: (a) temporal abstraction and (b) data abstraction. These thus represent the major topics of this section.

2.2.1 Related Work

This section focuses on the temporal and data abstraction ideas described in [Mel90] used in most of the microprocessor verifications cited earlier in this report.

Temporal Abstraction

Temporal abstraction relates the coarse-grained time at an abstract level of description to a fine-grained concrete time. The objective of a temporal abstraction approach is, therefore, to define an appropriate mapping between the two levels. This is necessary so that the state, input, and output signals defined over these times can be related.

In typical interpreter applications, the temporal relationship is defined with respect to the concrete-level signals. An 'event predicate,' **G rep**, shown among the following HOL expressions, when true, marks an 'instruction boundary event.' These events define the concrete times marking the boundaries of the abstract operations.

Event Predicate:	**(G rep) :time'→bool**
Instruction Boundary Event:	**G rep t' = T**

Numerous examples of event predicates exist. In the low-level modeling work of [Her88] and [Mel90], for example, clock-level operation boundaries are defined by the rising edge of the system clock. In the microprocessor verifications cited previously, the boundaries of the abstract-level assembly instructions are defined by the return of the microcode program counter to address zero.

Having an event predicate is not sufficient, however, to relate an abstract level to an underlying concrete level. For example, while the event predicate **G rep** defines the concrete times marking the abstract-operation boundaries, it cannot identify the *particular* abstract times that correspond to these boundaries. The usual approach to address this is to define the abstract time as the accumulated 'count' of the boundary events.

'Operation boundary predicates' implement this idea, as shown in the following HOL code. **NTH_TIME_TRUE t (G rep) 0 t'** relates the abstract and concrete times through the event predicate **G rep**. The predicate is read "**G rep** is true for the **t**'th time at concrete time **t'**." The 'temporal abstraction function,' **t_abs**, maps abstract time to concrete time using the instruction boundary predicate. For each abstract time **t**, it returns a concrete time, **t'**, such that **G rep** is true for the **t**'th time at **t'**.

Instruction Boundary Predicate:	**NTH_TIME_TRUE t (G rep) 0 t'**
Temporal Abstraction Function:	**t_abs :time→time' = λ t . ε t' . NTH_TIME_TRUE t (G rep) 0 t'**

Data Abstraction

Data abstraction relates the abstract-level data values to the data of the implementation. It is typically implemented with functions that map concrete values to abstract values. The following expressions show the abstraction functions mapping the state, environment, and output. While these functions can theoretically handle arbitrary mappings, in practice the abstract data structures are usually simple subsets of the underlying concrete data structures. Note that the types for these functions are different from those of the

generic functions shown at the beginning of Section 2. The generic functions apply to the data abstraction described in the next section.

Data Abstraction Functions

State Abstraction:	**s_abs :*state'→*state**
Environment Abstraction:	**e_abs :*env'→*env**
Output Abstraction:	**p_abs :*out'→*out**

Putting the Two Together

The following HOL expressions show how the temporal and data abstraction operators are normally combined to implement abstraction (**o** is the function composition operator). The abstract state signal **s**, for example, is defined as the expression within the parentheses on the right-hand side. The value defined by this expression applied to an abstract time **t** is seen to be a concrete value (**s' t'**) mapped through **s_abs**. The concrete time **t'** is the abstract time **t** mapped through **t_abs**.

Traditional Interpreter Abstraction Relationships:

```
s t = (s_abs o s' o t_abs) t
e t = (e_abs o e' o t_abs) t
p t = (p_abs o p' o t_abs) t
```

2.2.2 HPL Abstraction

The HPL approach to interpreter abstraction uses the same temporal abstraction ideas as above but otherwise differs in two fundamental ways: (a) it uses more-powerful data abstraction functions to implement interval abstraction (described in the Task 10 Specification Report [Fur93a]) and (b) it implements the abstraction within 'abstraction predicates' rather than embedding it within the interpreter correctness statement.

The following definition describes a typical abstraction predicate. Again, the temporal mapping uses the function **t_abs** defined above. The difference from the traditional interpreter approach is that the state abstraction functions are strengthened to support interval abstraction. For example, the state abstraction is implemented using **SAbs rep**, which operates on the concrete state *signal*, **s'** (type **:time'→*state'**), rather than the state *value*, **s' t'** (type **:*state'**). This gives the user of the abstraction function freedom to map multiple (temporal) instances of a concrete signal to the abstract level, and allows mapping from concrete times other than **t'**.

The abstraction is defined within an abstraction predicate, in contrast to current practice where it is defined within the interpreter correctness statement. One of the benefits of this approach is that abstraction is now defined and reasoned about as a stand-alone entity. This more closely matches our intuitive understanding of abstraction, which we view as defining the relationships between abstract and concrete variables, period. We do not require the existence of a correctness statement to think about these relationships. We also find the explicit naming of both the concrete and abstract signals to increase the clarity of these relationships. The implicitly defined abstract signals in the traditional approach provide a more cryptic definition.

HPL Abstraction Predicate:

```
|- INTRP_ABS rep s e p s' e' p' =
     ∀t. let t' = t_abs t in
           ((s t = SAbs rep s' e' p' t') ∧
            (e t = EAbs rep s' e' p' t') ∧
            (p t = PAbs rep s' e' p' t'))
```

As explained in [Fur93a], the traditional approach to interpreter abstraction, implemented in the GIT, is not flexible enough to handle the required mapping between the clock level and the transaction level of the PIU. This was our original motivation for examining alternative approaches, leading to the selection and further development of HPL.

2.3 Interpreter Liveness

Interpreter liveness is a subtle issue, and the reason for this is that it involves the Hilbert choice operator, ε. To help clarify things we consider again the HPL abstraction predicate, but we define it in a somewhat different way (and without ε) as shown in the following HOL code. This definition is actually one that could be used for an implementation proof (it has certain disadvantages with respect to composition however). We show it because it demonstrates the clear need to establish that **NTH_TIME_TRUE t (G rep) 0 t'** is true for the temporal variables **t** and **t'**. This is the interpreter liveness proof obligation.

Modified Abstraction Predicate:

```
⊢ INTRP_ABS_X rep s e p s' e' p' =
    ∀ t t'. NTH_TIME_TRUE t (G rep) 0 t'
        ⊃ ((s t = SAbs rep s' e' p' t') ∧
           (e t = EAbs rep s' e' p' t') ∧
           (p t = PAbs rep s' e' p' t'))
```

In the actual abstraction predicate (of Section 2.2.2) the concrete time is mapped from the abstract time using the rewritten **t_abs** function as: **t' = ε t' . NTH_TIME_TRUE t (G rep) 0 t'**. What can we say about this time **t'** ? Nothing, actually, unless we can prove the following subgoal: **∃ t' . NTH_TIME_TRUE t (G rep) 0 t'** In other words, just as above, to achieve an interpreter correctness proof, the predicate **NTH_TIME_TRUE t (G rep) 0 t'** must be proven true for the abstract instruction executed at time **t**. Without this the concrete and abstract variables cannot be related through the data abstraction functions. This proof obligation is easily obscured by the way that the abstraction predicate is defined in Section 2.2.2.

2.3.1 Related Work

In Melham's treatment of abstraction, the interpreter liveness proof obligation is to establish a 'universal liveness' property similar to the one shown next. Proving liveness therefore entails showing that an abstract-to-concrete temporal mapping exists for *all* abstract times (or **G rep** is true infinitely often).

Universal Liveness:

```
⊢ INTRP_LIVE rep =
    ∀ t. ∃ t'. NTH_TIME_TRUE t (G rep) 0 t'
```

When the event predicate depends on the interpreter input (i.e., **G rep = G' e' rep**) then universal liveness cannot be proven. The source of the input may simply stop transmitting, for example. In their low-level hardware verification work, both Melham and Herbert simply assumed universal liveness for the clock input signals within their event predicates [Mel90] [Her88].

When the event predicate is a function of the state (i.e., **G rep = G' s' rep**) then the situation is much improved, and universal liveness can be proven. In the microprocessor verifications of Graham and Joyce, for example, the liveness property proven was actually somewhat different from the above definition. The basic idea in their work was to instead prove the following two properties: (1) **G rep** is true in at least one state, and (2) whenever **G rep** is true in a state then it is true in some later state [Gra92] [Joy90]. The proofs

of these two facts are more easily accommodated in an interpreter correctness proof than a direct assault on the above definition. Windley's GIT incorporated many of these proof steps as part of the theory's infrastructure [Win90].

2.3.2 HPL Interpreter Liveness

For the vast majority of the transaction-level implementation proofs within the PMM, the event predicate is a function of the input rather than the state. The option of assuming universal liveness is not very appealing here, particularly for the *S* process that is concerned with start-up behavior. Here we would expect the event predicate (an active reset) to occur exactly once.

Fortunately, it is not really necessary to establish universal liveness to achieve an implementation proof. The sufficient property to establish is that, given that an abstract operation is executed at time **t**, then there exists a **t'**th occurrence of the event predicate. This property is defined next.

Conditional Liveness:

```
⊢ COND_INTRP_LIVE rep s e p =
    ∀ k t. EXEC rep k s e p t
        ⊃ ∃ t'. NTH_TIME_TRUE t (G rep) 0 t'
```

We have demonstrated in our on-going P-Port verification that this definition is strong enough to achieve implementation correctness proofs. It is also weak enough so that it can be proven itself, provided that the execution predicate and abstraction predicate are defined appropriately. We use an example from the PIU P-Port to illustrate what we mean here. More explanation of the following terms can be found in [Fur93a].

In the P-Port the event predicate is defined with respect to two L-Bus signals received from the local processor: **G rep** = λ **t'.** ¬ **BSel(L_ads_E (e' t'))** ∧ **BSel (L_den_E (e' t'))**. Now, turning to the transaction level, the execution predicate **EXEC rep k s e p t** is true if one of the six valid L-Bus transaction opcodes is received at time **t**; **PBM_WrLM, PBM_WrPIU, PBM_WrCB, PBM_RdLM, PBM_RdPIU,** or **PBM_RdCB.** In order to be able to prove conditional liveness, we place the following constraint on the abstraction: We require that the abstraction function define a transaction opcode equal to one of these values only if **NTH_TIME_TRUE t (G rep) 0 t'** is true, where **t'** is defined in the normal way as **t_abs t.**

This constraint says that we don't consider the P-Port to have received a valid transaction opcode at time **t** unless the P-Port has received **t** (clock-level) transaction requests. This is really not much of a constraint since we would expect this fact to be true anyway.

To conclude this section, we point out only that it is the separation of the operation decoding function into its own execution predicate that permits this preferred solution to the interpreter liveness problem.

2.4 Interpreter Correctness

With a few exceptions (e.g., [Hun86]), interpreter correctness statements have generally followed the same form. These statements define an implication in which the behavior defined by the implementation interpreter implies the behavior defined by the specification interpreter. If we ignore the issues of abstraction and liveness, these statements are represented by the following pseudo-code.

General (Stripped Down) Interpreter Correctness Definition:

```
⊢ imp_intrp behavior ⊃ spec_intrp behavior
```

Proving correctness statements such as this directly is usually avoided because of the complicated behaviors involved. One common approach used to break these proofs into smaller subgoals is to perform

a case split on the abstract-level instruction set. When this is done the proof obligations resemble the following pseudo-code. Each of these subgoals is read: "if we assume the concrete interpreter behavior, and if we assume that the condition for (abstract) operation J's execution is satisfied, then the specified behavior for operation J results."

General (Stripped Down) Operation Case Split:

$\vdash$ *imp_intrp behavior* $\supset$ *spec_op1 execution condition* $\supset$ *spec_op1 postcondition*

⋮

$\vdash$ *imp_intrp behavior* $\supset$ *spec_opN execution condition* $\supset$ *spec_opN postcondition*

2.4.1 Related Work

The microprocessor verifications cited earlier employ a correctness statement similar to the one shown next. This statement says: "if we assume the given implementation and interpreter liveness, then the instance of the specification interpreter 'defined by the abstraction' is correct." This embedding of the abstraction within the correctness statement follows [Mel90], and is the approach used to formally link the abstract- and concrete-level variables.

Interpreter Correctness Definition (based on [Mel90],[Joy90], etc.):

```
∀ rep s' e' p'.
 INTRP_imp rep s' e' p' ∧
 INTRP_LIVE rep
 ⊃ INTRP_spec rep (s_abs o s' o t_abs)
                  (e_abs o e' o t_abs)
                  (p_abs o p' o t_abs)
```

Given this basic form, the correctness proof can be approached in a couple of ways. The Viper, Tamarack, and SECD proofs all employed variations of a common technique. Windley's GIT-based verification of AVM-1 was a significant departure from this however.

'FSM' Approach

As indicated in Section 2.1, the FSM-style interpreter specification approach presents some difficulties for correctness proofs. The problem is that the embedding of the instruction decoding within the **NEXT_STATE** function makes it somewhat difficult to perform the desired case split on the instruction set. Although this is not a major problem, it does require extra work in performing the verification.

Joyce, in the proof of Tamarack [Joy90], handled the instruction case split by explicitly defining a proof obligation for each instruction in terms of its 3-bit opcode bit combination. In all, there were eight instructions verified this way. Some final processing was needed to show that the collection of cases proved actually covered all possible combinations of bit patterns.

In the SECD proof [Gra92], Graham introduced a disjunctive 'valid_program_constraint' as a precondition within the correctness statement. Each disjunct represented the conditions under which that particular instuction was selected for execution. The case split was performed directly as part of the proof, with each case using its disjunct as an assumption. A disadvantage of this approach was the extra work required to specify the constraint, which was lengthy and quite detailed.

GIT Approach

Windley's work on the GIT recognized that the steps needed (in [Joy90]) to process the individual instruction proofs into a finished verification were independent of the actual system being verified. Rather, they were functions of the structure of the interpreter model, which could be fixed. Where the GIT takes a significant departure from these previous efforts is in collecting up these sorts of proof requirements and proving them within a generic theory.

This has two beneficial aspects. First of all, the workload for the user of the theory is reduced since the theory proofs are reused. Secondly, the theory presents to the user a clear definition of the user's proof obligations. For example, the GIT presents the following two obligations to the user.

GIT User Proof Obligations [Win90]:

```
(∀ s' e' p' k . INST_CORRECT s' e' p' k) ∧
(∀ s' e' p' k . OUTPUT_CORRECT s' e' p' k)
```

We don't go into the details of **INST_CORRECT** and **OUTPUT_CORRECT** here since they are somewhat lengthy – these details can be found in [Win92]. The important point to be noted about these obligations is that they both contain the 'for all **k**' that makes clear the need for an instruction case split. Again, the proof infrastructure within the theory takes care of the details necessary to derive the interpreter correctness statement described previously.

2.4.2 HPL Correctness

HPL interpreter correctness is proven using the same general approach described above. There are some differences however, primarily in (a) the handling of abstraction and (b) the support for instruction case splitting.

The HPL interpreter correctness definition is shown next. This statement says: "if we assume the given implementation and interpreter liveness, and if the abstraction is defined according to the abstraction predicate there, then the specification interpreter is correct."

HPL Interpreter Correctness Definition:

```
∀ rep s e p s' e' p'.
INTRP_imp rep s' e' p' ∧
INTRP_ABS rep s e p s' e' p' ∧
COND_INTRP_LIVE rep s e p
⊃ INTRP_spec rep s e p
```

We view this correctness statement as preferable over the statement of the last section in two respects: (a) the theorem consequence contains the specification interpreter, rather than a particular instance of the interpreter, and (b) abstraction is clearly (and correctly) treated as an *asserted* entity by its very position within the correctness statement. The practical differences between the two definitions may be small however.

Instruction case-split support is not evident from this definition. To see where it exists we need to revisit the interpreter definition described in Section 2.1.2. Informally, this definition says: "for all **k** and **t**, if instruction **k** is executed at time **t** then the postcondition for **k** is true at **t**." Thus, the instruction case split requirement is clearly indicated in the interpreter definition itself. It is made visible during an interpreter proof as soon as the abstract interpreter is rewritten, as a normal part of the initial proof steps in any verification.

3 Issues in Secure Abstract-Level Composition

An important capability of a theorem-proving approach to hardware verification is the ability to achieve provably correct compositions of subsystem models at high levels of abstraction. This is an area in which theorem proving holds a potential advantage over competing approaches such as model checking [McM93], as well as others.

There are good practical reasons for composing at high levels of abstraction. As pointed out in the Task 10 Specification Report [Fur93a], abstract-level compositions are easier to verify than those performed lower in the hierarchy. In addition, the difficult implementation proof performed within a subsystem to achieve a verified abstract level is reused every time the subsystem is designed into a new system configuration. In contrast, composition proofs must be repeated for new configurations.

For the specification and verification of large hardware systems there is probably no choice but to perform as much abstraction as possible within subsystems, before composition, just to get a handle on the complexity. As demonstrated in the work under Task 10, performing the clock- to transaction-level abstraction within the PIU ports greatly reduced the complexity of the PIU system model, and is expected to reduce the amount of required theorem proving as well.

In this section we overview some important issues in achieving provably correct compositions of abstract subsystem models. We first review the role of abstraction in this process. Following this, we describe the role of temporal abstraction for secure composition at low levels of abstraction. We then contrast this to the temporal abstraction necessary for composition at levels comparable to the transaction level. The interface-event clocking introduced here provides the subject for Section 4.

3.1 The Role of Abstraction

The problem of ensuring the soundness of abstract-level composition is illustrated by Figure 3.1 At the top of this figure are two components, **f** and **g**, with output **a** and input **b**, respectively. Each of these components is an abstract representation of the component beneath it, **f'** and **g'**, respectively. In its intended operation, the abstract component **g** should receive the output of component **f**; in other words, we desire that the value on signal **b** be equal to the value on signal **a**, for all time **t**. The problem is to show that making this connection (composing **f** and **g**) is 'sound,' or that the composed abstract-level system remains an accurate abstraction of the composed concrete-level system.

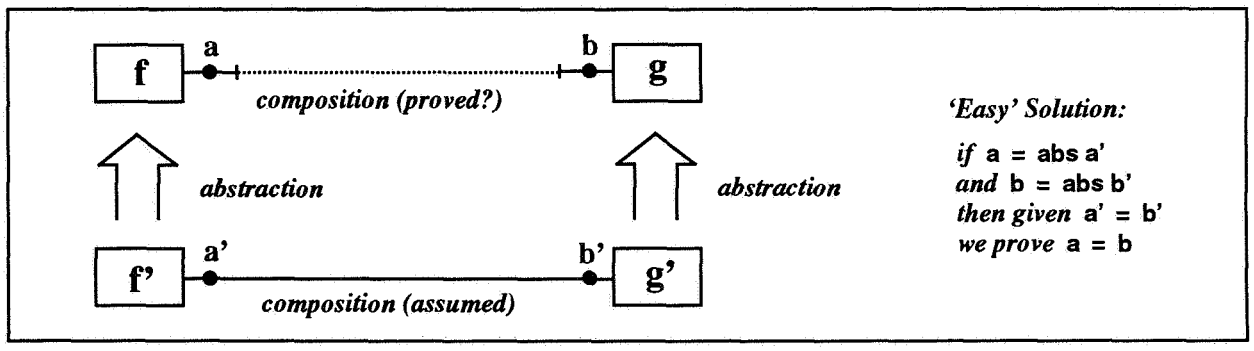


Figure 3.1: The Problem of Composing Abstract Models.

At the lowest level of abstraction in a model hierarchy, it is necessary to make the assumption that components are interconnected in some specified way. This is akin to assuming that the individual components have a specified behavior, and, hence, is unavoidable. However, at high levels of abstraction it is not acceptable to make such assumptions for the interconnections, just as it is not acceptable to assume the required

behavior of the individual components. Fortunately, we don't need to make these assumptions, since the necessary properties can, in principle, be proven from the underlying implementation.

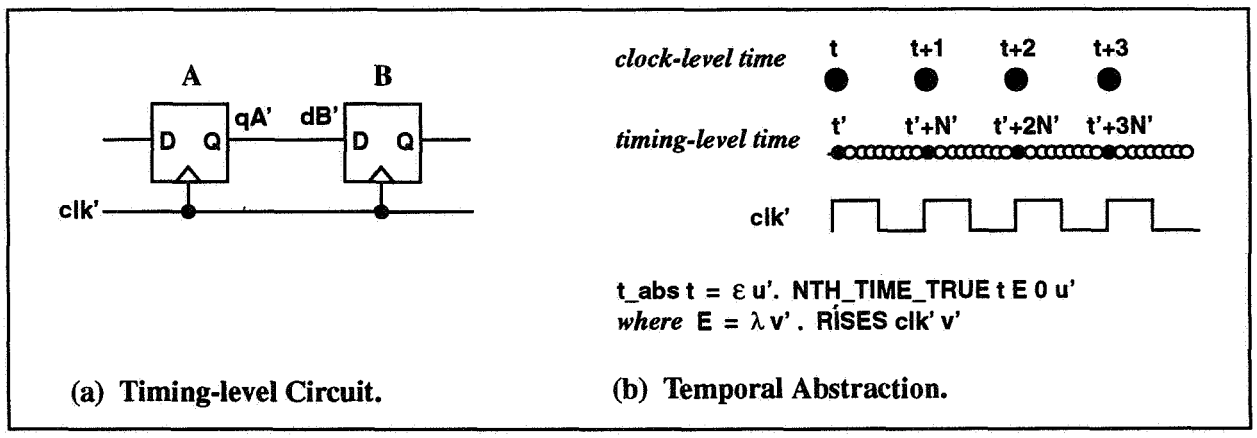
Referring again to Figure 3.1, the abstract-level composition proof obligation is to show that the signals **a** and **b** are equal, given the specified models for components **f'** and **g'**, plus their indicated interconnection, plus the abstraction linking **f'** with **f** and **g'** with **g**. The easiest way that this can be accomplished is if **a** and **b** are defined using a common abstraction function, **abs** for example, from their concrete counterparts **a'** and **b'** Since it is already assumed that **a'** and **b'** are equal, the desired property that **a** (= **abs a'**) and **b** (= **abs b'**) are equal would result directly.

In fact, the condition in which the abstract component inputs and outputs are abstractions of only the concrete inputs and outputs (and not state) holds among the transaction- and clock-level models of the PMM. The PMM specification hierarchy was constructed in this way partly in response to the composition problems that were anticipated during Task 10.

3.2 Temporal Abstraction for Globally-Clocked Systems

The work referred to in the last section (in [Her88] and [Mel90]) was concerned with compositions at the 'clock' level of abstraction, using concrete-level components defined at the 'timing' level. At the clock level, a time step corresponds to a single period of the system clock. The *timing level* [Her88] has as its time step a fixed increment of real time, such as a nanosecond or a fraction of a nanosecond. Even though we do not use a timing level in our own work, it is worthwhile to examine the temporal abstraction used in this prior work in preparation for the transaction-level discussion of the next section.

Figure 3.2(a) shows a very simple circuit that we use to illustrate some of these concepts. The flip-flops are both assumed to change their states on the rising edge of the clock signal **clk'**. The output of flip-flop A is assumed to be connected to the input of flip-flop B at the timing level of abstraction shown in the figure. The composition problem here is to prove that the same interconnection can be made at the clock level of abstraction.



(a) **Timing-level Circuit.** (b) **Temporal Abstraction.**

Figure 3.2: Temporal Abstraction Linking the Timing and Clock Levels.

Almost intuitively, hardware designers know that two clock-level flip-flop models can be interconnected this way, provided that the applicable setup and hold times are all met. Ignoring these details, which we consider to be data abstraction issues, the underlying reason why this is so is that both flip-flops are clocked on the same edge of the same clock. In other words, they both employ the same temporal abstraction to link their timing and clock levels. As shown in Figure 3.2(b), the temporal abstraction function **t_abs** relates the clock level to the timing level at the rising edges of the timing-level clock. The event signal **E**,

defined using the predicate **RISES**, is true precisely at these points of interest. **RISES clk' v'** is true precisely when **clk' v'** is true and **clk' (v'-1)** is false.

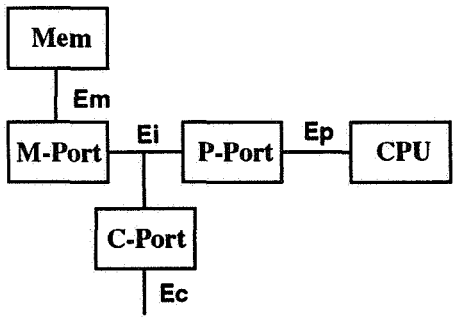
The synchronous timing control that is provided by a common clock is instrumental in making the design of even large digital systems tractable. Again, it is the use of a common temporal abstraction function implied by this that eases the composition-correctness burden on the digital designer. Likewise, this common clock simplifies the formal modeling and verification of composition at the clock level of abstraction. Unfortunately, from a modeling standpoint, digital designers do not include global clocks in their designs that step at the rate of all the levels in a specification hierarchy. The techniques described in this section need to be generalized to work at these higher levels.

3.3 Temporal Abstraction for Interface-Clocked Systems

In contrast to the low-level work described above, we are aware of no work to compose interpreters at higher levels of abstraction. The bulk of the work done by the theorem-proving community targets microprocessors. In this work the top level is typically the *macroinstruction level*, where a time step corresponds to the execution of an assembly-language instruction. The level immediately below this is often at the *microinstruction level*. When compositions are performed as part of these microprocessor verifications however (for example between the microprocessor and system memory), it is done at the clock level or lower (e.g., [Hun86],[Coh88],[Joy90],[Win90],[Gra92],[Lev93]). In [Fur93a] we explained why we reject this approach for the PMM.

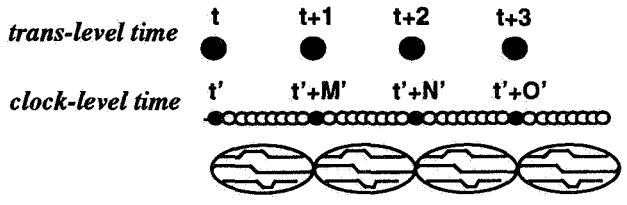
One difficulty with higher-level composition is that, in direct contrast to the clock level, there is no notion of a global clock to synchronize the actions of the participating subsystems. For this reason, higher-level composition requires a certain amount of flexibility in defining what constitutes a 'clock,' as well as its corresponding 'time steps.'

As before, one key to secure higher-level composition is the use of a consistent temporal abstraction for the subsystems being composed. For our transaction-level modeling, we have accomplished this by focusing on the bus protocols used by the subsystems of the PMM. At this level, a time step is defined by the occurrence of an event corresponding to the initiation of a new bus transaction. Figure 3.3 describes these events for the four buses of the PMM: the L-Bus (or P-Bus), I-Bus, M-Bus, and C-Bus.



Ep: *L-Bus event signal*
Ei: *I-Bus event signal*
Em: *M-Bus event signal*
Ec: *C-Bus event signal*

(a) PMM Structure and Event Signals.



t_abs t = ε u'. NTH_TIME_TRUE t E* 0 u'

Ep = λ t. ¬ L_ads_t ∧ L_den_t
Ei = λ t. ¬ l_male_t ∨ ¬ l_rale_t ∨ ¬ l_cale_t
Em = λ t. FALLS MB_cs_eeprom_t ∨
FALLS MB_cs_sram_t
Ec = λ t. ¬ CB_rqt_t

(b) Temporal Abstraction for the Four Buses.

Figure 3.3: Temporal Abstraction Linking the Clock and Transaction Levels.

The L-Bus event signal **Ep**, for example, when true for a clock-level time **tp'** indicates that a new L-Bus transaction has begun, and, hence, that the transaction-level time should be incremented.

In a system with multiple buses then, it is possible that multiple clocks will exist. Section 4 covers the issues involved in modeling and verifying interpreters with interface clocking. (In the rest of this report we generalize the notion of bus protocols to include any communication policy between two or more subsystems; we use the more generic term *interface protocol* to describe them.)

4 Synchronizing Interface-Clocked Interpreters

The last section has shown that interface-event clocking can present scenarios in which interpreters relate to different events over their input and output interfaces. Since these events dictate the temporal abstraction for the abstract level, this implies that the abstract-level temporal variables for an interpreter's input and output may themselves be different. In this section we examine how this possibility affects the modeling and verification of interpreter-based systems.

The discussions of this section will be largely example driven. Figure 4.1 introduces the modeling style that will be used throughout the section. The interpreter in this figure has an input event predicate **Fe rep ef** and output event predicate **Fp rep pf**. The 'representation' variable **rep** has the same interpretation as in the previous sections. For example, **Fe rep** is a generic operator that maps a signal, **ef** in this case, to the booleans. The actual definition of **Fe rep**'s behavior is undefined, only its type is known. Within the actual HOL segments shown in this section, the complete expression will be used. However, in the text we will usually employ the abbreviated form: **Fe**, **Fp**, etc.

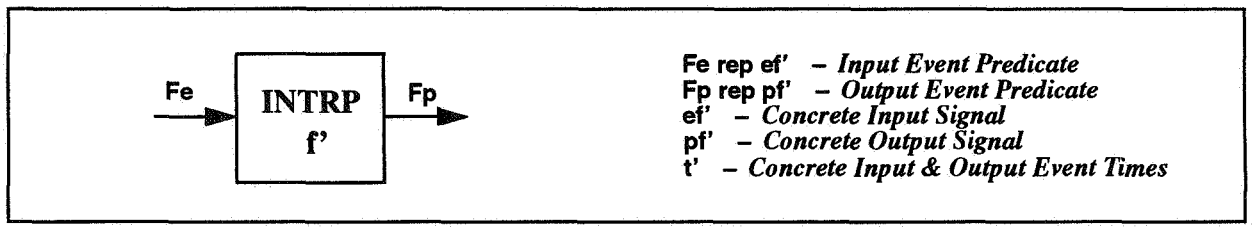


Figure 4.1: Description of an Interpreter's Transaction Events and Concrete-Level Variables.

The input and output signal names follow a standard pattern of prefixing an **e** or a **p**, for the inputs and outputs respectively, to the name of the interpreter under consideration. The concrete-level signal names **ef'** and **pf'** are therefore used for the concrete-level intepreter **f'** All concrete signals used in the following examples are of the type **":time'→*io"**; the abstract signals map abstract time, **t**, to the same signal-value type ***io**.

To permit this section to focus on issues of composition, the models in this section avoid unnecessary complexity wherever feasible. For example, the behavior of both concrete- and abstract-level interpreters is a simple flowthrough: **∀ t'. pf' t' = ef' t'** in the above example. We will describe other assumptions as they are needed.

Moving on to the interpreter modeling issues themselves, we begin by pointing out that, even though a given interpreter's event predicates may be different for the inputs and outputs, this does not mean that the temporal variables themselves must be different. A good illustration of this is the P-Port within the PIU. Here, the port's input events are defined over the L-Bus and its output events over the PIU I-Bus. These are clearly different events yet the transaction-level temporal variable is the same for the inputs and outputs. The reason for this is that the input-to-output event mapping is one-to-one and onto. In other words, each input event causes at least one output event, and, furthermore, it causes at most one output event.

Therefore, an important consideration for interface-clocked interpreters is the relationship between the occurrences of the input-transaction requests and the resulting output-transaction requests. These requests may or may not be one-to-one and they may or may not be onto.

To gain a better picture of exactly what we mean here, Figure 4.2 shows these two relationships graphically. The one-to-one mapping, defined by the HOL definition **TRANS_ONE_TO_ONE** in part (a), describes an interpreter where every input transaction event results in a unique output transaction event. In other words, if an input request arrives at concrete time **te'**, then no further requests will arrive until after an output

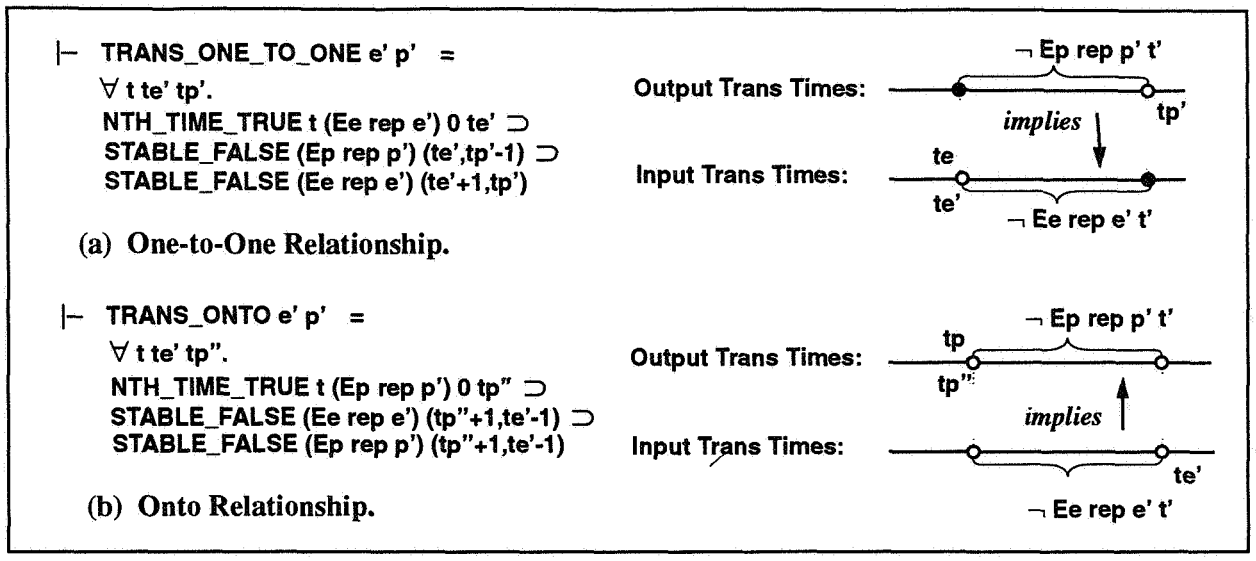


Figure 4.2: Transaction One-to-One and Onto Relationships.

transaction is begun. The predicate **STABLE_FALSE f (t1,t2)** is true precisely when **f t** is **F** for all **t** in the closed interval [**t1**, **t2**].

The onto mapping defined in part (b) describes an interpreter where every output request is in response to a unique input transaction event. Here, if an output request is transmitted by the interpreter at concrete time **tp"**, then no more such requests will be transmitted until a new input transaction request is received.

The one-to-one and onto relationships just described provide for a four-element partitioning of the class of (single-input, single-output) interface-clocked interpreters:

(1) Interpreter transactions are *one-to-one* and *onto.*
(2) Interpreter transactions are *one-to-one* but not onto.
(3) Interpreter transactions are not one-to-one but are *onto.*
(4) Interpreter transactions are neither one-to-one nor onto.

Examples of all four interpreter types can be found in the PMM, depending on how one defines the transaction events for the various buses. As we have defined them, they lead to the existence of interpreter types (1), (2), and (3). In the following three subsections we examine each of the first three interpreter types, primarily through simplified examples.

4.1 Bijective Monorate Interpreters

When the mapping between the interpreter input transactions and the output transactions is bijective (both one-to-one and onto) we have the familiar situation in which the abstract clock rates are the same for the interpreter inputs and outputs. In order to introduce some concreteness into the following discussion, we will refer often to the simple circuit example shown in Figure 4.3.

One-to-one and Onto Conditions Satisfied:

(one-to-one)
∀ t'. Fe rep ef' t' ⊃ Fp rep pf' t'

(onto)
∀ t'. Fp rep pf' t' ⊃ Fe rep ef' t'

Concrete Interpreter Behavior:
⊢ INTRP_f' ef' pf' = ∀ t'. pf' t' = ef' t'

Abstract Interpreter Behavior:
⊢ INTRP_f ef pf = ∀ t. pf t = ef t

Input Abstraction:
let t' = ε t'. NTH_TIME_TRUE t (Fe rep ef') 0 t' in
ef t = ef' t'

Output Abstraction:
let t' = ε t'. NTH_TIME_TRUE t (Fp rep pf') 0 t' in
pf t = pf' t'

Figure 4.3: Example Bijective Monorate Interpreter Model.

The synchronization problem for monorate interpreters is fundamentally different from the intepreter synchronization problems of the following two sections. Here, we are concerned with an aspect of the interpreter implementation verification. In a verification such as this it is clearly necessary to establish a proper input-output relationship for the concrete-level interpreter; in other words, in response to an input event request at time t' the interpreter must be shown to produce correct outputs. Now, having this, it is an additional step to prove that the t'th input request produces the correct behavior corresponding to the t'th output transaction, and not some other output transaction.

A fair amount of our Task 10 work dealt with this synchronization problem for the PIU P-Port, and it resulted in the solution described in [Fur93a]. In our work under Task 12 we have revisited the Task 10 solution and made a fairly significant improvement to it. We don't detail the Task 10 approach here except to say that it requires two additional theorems not required by the new approach that we describe next.

The Task 10 work taught us that synchronizing an interpreter's abstract-level inputs and outputs can be divided into two subproblems that can be handled separately. These are: (a) establishing interpreter 'output liveness', and then, from this, (b) establishing the correct 'transaction ordering.' The following two HOL definitions precisely capture these two concepts. In these definitions, to save space we have introduced the terms ***Ee*** and ***Ep*** to represent **Ee rep e'** and **Ep rep p'**, respectively.

Interpreter Output Liveness:

⊢ OUT_TRANS_LIVE e' p' =
∀ t te'.
NTH_TIME_TRUE t *Ee* 0 te' ⊃
∃ tp'. STABLE_FALSE_THEN_TRUE *Ep* (te',tp')

Interpreter Trans Ordering:

⊢ NEXT_OUT_TRANS_IS_NTH e' p' =
∀ t te' tp'.
NTH_TIME_TRUE t *Ee* 0 te' ⊃
STABLE_FALSE_THEN_TRUE *Ep* (te',tp') ⊃
NTH_TIME_TRUE t *Ep* 0 tp'

The predicate **OUT_TRANS_LIVE** is true precisely when an output transaction event exists following every input transaction event. This is a property that clearly must be proved, since otherwise there would be no hope of establishing a proper temporal mapping on the interpreter output side. It is also a property that should be provable directly from the concrete-level definition of the circuit under consideration. Nothing is said about the ordering of such an output request, only that one must exist sometime on or after the arrival of the input request. The on-going P-Port verification includes this proof.

The predicate **NEXT_OUT_TRANS_IS_NTH** is distinguished from **OUT_TRANS_LIVE** in that it is usually not possible to be proved directly from the concrete-level circuit. The reason for this is that it involves the abstract-level time **t**. Since circuits don't normally maintain a memory of the number of input and output transactions processed, there is no notion of **t** contained within the circuit description. These types of theorems are most naturally proven by induction; in this case, on the abstract time **t**.

The following HOL segment describes an approach to achieving a proof for **NEXT_OUT_TRANS_IS_NTH**. In addition to the definition **OUT_TRANS_LIVE** just discussed, proof obligations exist for the definitions **TRANS_ONE_TO_ONE**, **TRANS_ONTO**, and **FIRST_TRANS_CAUSAL**. The first two of these were defined at the beginning of Section 4; the third is shown here. **FIRST_TRANS_CAUSAL** says that no output transactions will be produced before the first input transaction is received. Having proofs for the four theorem obligations, the justification theorem provides the desired result. The HOL listings in Appendix B contain these definitions and the proof of the justification theorem.

'Interpreter Trans Ordering' Proof Obligations:

```
|- ∀ e' p'. TRANS_ONE_TO_ONE e' p'
|- ∀ e' p'. TRANS_ONTO e' p'
|- ∀ e' p'. FIRST_TRANS_CAUSAL e' p'
```

where:

```
|- FIRST_TRANS_CAUSAL e' p' =
     ∀ te'.
     STABLE_FALSE (Ee rep e') (0,te'-1) ⊃
       STABLE_FALSE (Ep rep p') (0,te'-1)
```

Justification Theorem:

```
|- ∀ e' p'.
   TRANS_ONE_TO_ONE e' p'  ∧
   TRANS_ONTO e' p'  ∧
   FIRST_TRANS_CAUSAL e' p'  ∧
   OUT_TRANS_LIVE e' p'
   ⊃ NEXT_OUT_TRANS_IS_NTH e' p'
```

In the tradition of Windley's generic interpreter theory, we believe that the breakdown of theorem proving tasks into two distinct classes, as shown here, has a great practical benefit in reducing the number of theorem proving tasks that must be repeated in every interpreter verification. The advantage to proving transaction ordering this way is that the justification-theorem proof could be embedded within a generic theory and then simply reused whenever a new circuit is being verified. The user of the theory would therefore not need to be concerned with constructing the proof. In general, it would be desirable to include into such a theory all of the proof infrastructure that doesn't directly involve the circuit implementation or specification. In this way, the user would only be concerned with proving facts that were specific to the particular circuit being addressed.

We are not proposing to develop a generic theory for interface-clocked interpreters in the near term, since a significant amount of basic research remains to be done; however, we believe that it is a good practice to structure theorem-proving tasks so that they could be easily incorporated into a future theory.

4.2 Injective Multirate Interpreters

The monorate interpreters addressed in the last section resemble very closely the interpreters employing global clocks, in the sense that the output times and the input times were the same for each state of the machine. Of course, there is no inherent reason that an interpreter's input and output transaction events must be one-to-one and onto. When they are not, then the input and output times can be different, even for those corresponding to a common clock cycle. This possibility can seem strange unless one keeps in mind the fact that the *interface events* dictate the associated clock rates, and that these events can occur at different rates. As we shall explain in this section, the benefit that is gained by handling interpreters this way is provably secure interpreter composition. And this composition is performed at a high level of abstraction, such as the PMM transaction level.

In this section we address interpreters where the mapping between input and output transaction events is one-to-one but not onto. In other words, for every input event there are one or more output events. Figure 4.4 depicts this scenario, using a version of the simplified example used in the last section. Again, the output behavior of this interpreter, at the abstract level, is the identity function, as are the data abstractions for the input and output variables.

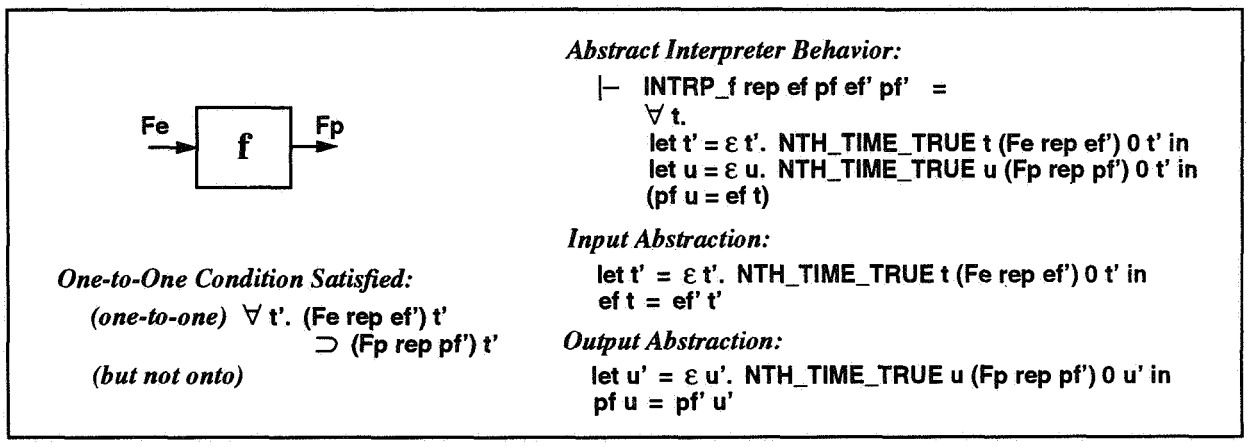


Figure 4.4: Example Injective Multirate Interpreter Model.

The temporal abstraction functions in this figure clearly illustrate the multirate aspect of this example interpreter. Abstract time **t** at the interpreter input corresponds to the concrete time **t'**, where event **Fe** is true for the **t**'th time. Likewise, abstract time **u** at the output marks the concrete time that **Fp** is true for the **u**'th time. For this interpreter the relationship $t \leq u$ holds.

The interpreter behavioral model has some unfortunate complexity. Our objective at the start of this task was to employ the simpler relationship ∀ **t**. ∃ **u**. **pf u = ef t** to express interpreter behavior. However, this was not satisfactory. While it was possible to successfully verify the behavior with respect to a concrete-level implementation, using it to verify behavior higher up in the hierarchy proved to be extremely difficult. We were not able to accomplish this, due to the existential quantifier's inability to 'keep track' of the necessary relationship with the concrete-level time. More discussion on this general topic proceeds below.

The other approach that we considered was the relationship ∀ **t u. pf u = ef t**. This also failed to work, but for the opposite reason; that is, it was too strong a statement to verify with respect to the implementation. Specifying the abstract time **u** for the interpreter output has thus turned out to be a fine balancing act between constraining it too much and not constraining it enough. Although somewhat ugly, the interpreter definition in Figure 4.4 achieves the proper balance.

In this interpreter definition the output expression contains the abstract time **u**, defined as the 'event count' corresponding to the concrete time **t'**. The definition of **t'** here is critical – it is defined as the concrete time that the input event is true for the **t**'th time. Thus, the **t**'th input event and **u**'th output event are related through their common concrete-level time **t'**.

To provide further explanation of these types of interpreters and to answer the question: "yes, but do they really exist in practice?" we now present a fairly substantial example. This example is actually a simplified version of certain ports within the PIU. It can also be seen to represent an important class of computing structures, namely shared-memory systems. All of the definitions and proofs shown here have been implemented in HOL and are contained in Appendix B.

Figure 4.5 shows the example circuit. The components labeled **f'**, **g'**, and **h'** are intended to mimic certain aspects of the behavior of the PIU C-Port, P-Port, and M-Port, respectively, while **i'** represents the I-Bus. The individual concrete-interpreter behaviors are simply flowthrough. The **i'** component behavior is viewed in a highly simplified way to avoid complexity not relevant to the immediate discussion. We assume that a transaction request from either of the source components, **f'** or **g'**, is passed on to the **h'** component. Thus, we are ignoring possible bus contention here.

The abstract interpreter behavior is also a simple flowthrough for each of the three interpreters. In this figure, and in the following discussion, it is important to keep in mind that the abstract time variables (**t** here) are defined over the interface events shown in the appropriate figure. For example, **t** within the **INTRP_f** model counts the input events **Fe** and the output events **Fp**.

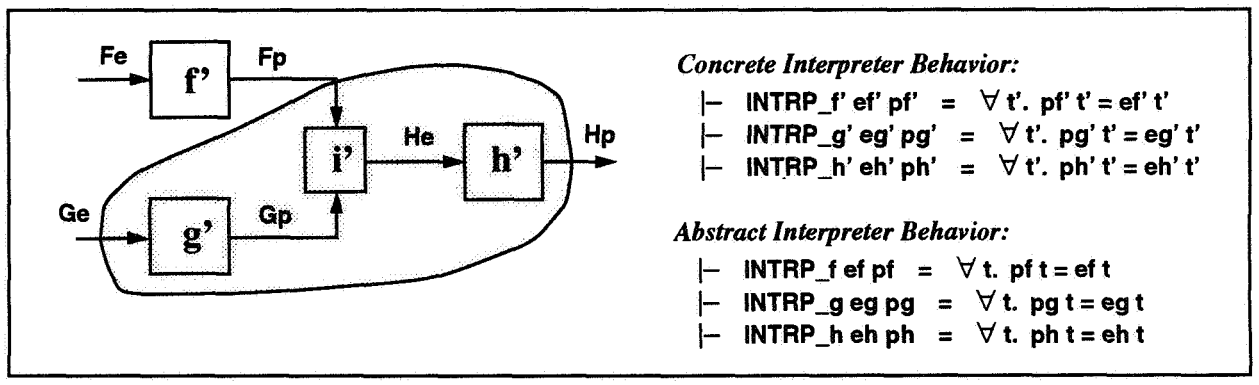


Figure 4.5: Example System Exhibiting Injective Multirate Behavior.

So far we have seen nothing but one-to-one and onto behavior within these interpreters. Where we lose the onto behavior is in going to a higher level of abstraction. For example, in a direct parallel to the *P* Process of the PIU transaction-level specification, consider the behavior of a composite system containing the components **g'**, **h'**, and **i'**. Input events to this system are those denoted by **Ge** (analogous to PIU L-Bus requests) and the output events are represented by **Hp** (analogous to M-Bus output requests). In this composite system the input and output events clearly proceed at different rates, since the output events **Hp** are also caused by the events **Fe**; not just the input events **Ge**.

In light of the multirate interpreter model of Figure 4.4, it is perhaps an understatement to say that having multiple clocks is not desirable for a top-level specification. For our small system example then, the interpreter model shown in Figure 4.6 is preferred over the earlier multirate model style of Figure 4.4. In the composite system model here, the input and output values are both defined with respect to the input-event clock. In the rest of this section we explain how such a behavioral model can be verified with respect to the multicomponent configuration of Figure 4.5.

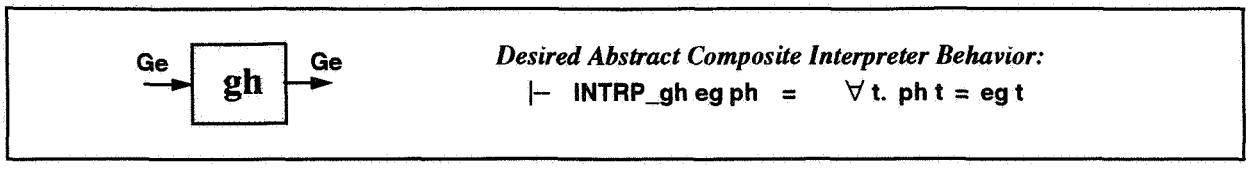


Figure 4.6: Desired Composite-System Behavior for System Example of Figure 4.5.

Figure 4.7 describes an implementation hierarchy linking the concrete models of Figure 4.5 to the abstract model just described. To simplify things somewhat, we are ignoring the intermediate bus i', and therefore we assume that the components **g'** and **h'** are connected.

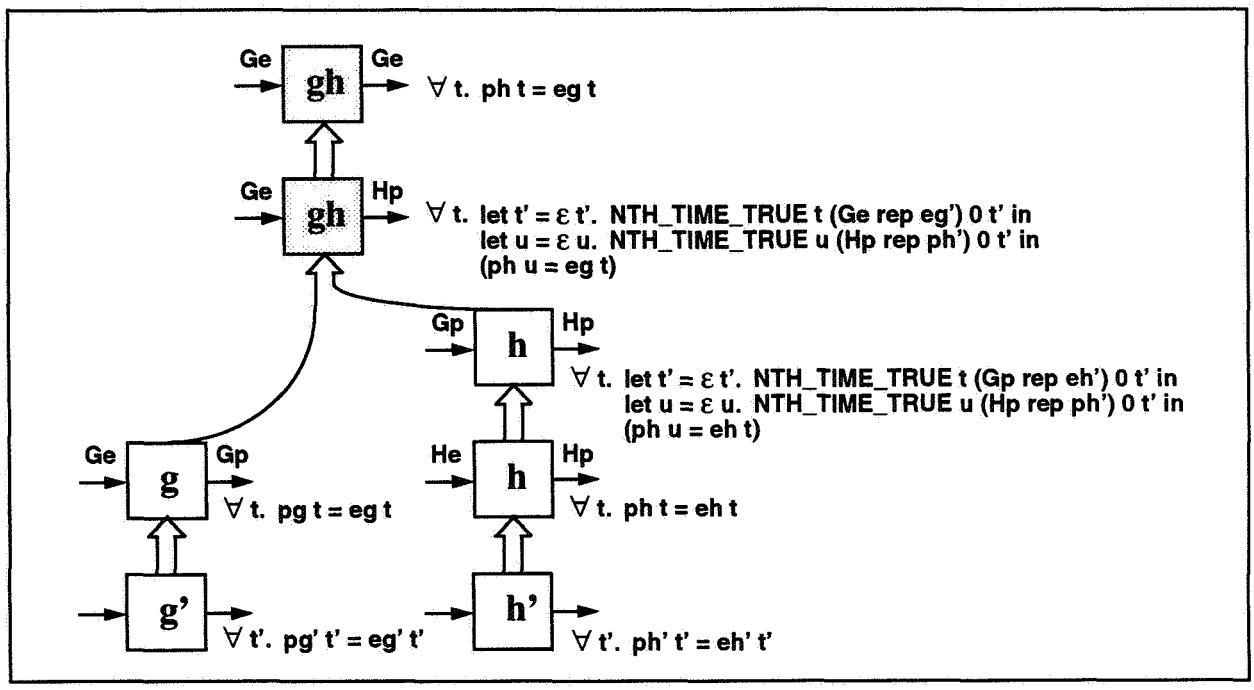


Figure 4.7: Specification Hierarchy for Circuit of Figure 4.5.

At the bottom of the hierarchy are the concrete-level models with their associated behaviors shown at the side. Immediately above these are the abstract-level models for the same components. Here, the input and output values shown in the equations are based on the event predicates shown in the figure. An implementation proof linking these two levels of models would use the techniques described in the Task 10 report [Fur93b]. In our HOL development in Appendix B we assume the implementation proofs for **g** and **h**, to more quickly get on with the issues of composition.

Having verified models at the second level of this hierarchy, we would now like to compose them into a composite '**gh**' system. Unfortunately, we cannot do this directly because of the mismatch in temporal abstraction between **g** and **h**, as evidenced by the different 'event counters' **Gp** and **He**. The solution to this dilemma is to redefine the abstraction for the **h** input. Instead of counting the events **He**, we now count **Gp**. This produces the complicated interpreter expression for **h** shown in the figure.

The proof verifying that this new model is a valid abstraction of the old one is fairly straightforward. The key to making it work is the fact that a common concrete **t'** exists for the **t'**th **Gp** event and the **u**'th **He** event.

With common temporal abstractions defined for the interfacing signals, the components **g** and **h** are then composed to form a new structural model. The proof for this step required some unexpected work to ensure that the intermediate concrete-level signals were available to several of the necessary definitions. Normally these signals are discarded at the earliest possible time.

From the structural model, a behavioral abstraction was performed to create the model **gh** shown in the figure. As in the typical stucture-to-behavior proofs within the PMM, this proof was straightforward and short.

The model verified in this step still maintains the complex multirate behavior of the component **h'** In fact, had we desired, we could have avoided this scenario altogether by simply making **h**'s output count at the rate **Gp** to begin with. Our reason for not doing this was to remain faithful to the PMM design. In this design, the output of **h** mimics the M-Port value transmitted over the PMM M-Bus. In order to compose the M-Port (hence the PIU) with the local memory we will require a common temporal abstraction for this interface. Thus we keep **Hp** to demonstrate that this composition can be done.

The final step in the proof chain is to redefine the abstraction for the system **gh** as a sort of inverse to the operation done for the **h** component. In fact, the proof for this verification is remarkably similar to that developed for **h**. Again all of these definitions and proofs are contained in Appendix B.

4.3 Surjective Multirate Interpreters

In this section we address the reverse scenario from that of the last section, namely, interpreters where the input-to-output transaction mapping is onto but not one-to-one. In this case, for every interpreter *output* event there are one or more input events. Another way of looking at this is that only a subset of the input events cause output events.

Figure 4.8 shows an example of this type of interpreter. All of the assumptions and simplifications used in the previous sections are applied to this example, as well as those that follow in this section. In this particular figure, we note that the interpreter behavior is similar to that of the injective interpreter example in Figure 4.4. In fact, the temporal abstraction is identical. The only difference is the addition of the conditioning by **F rep (ef t)** here.

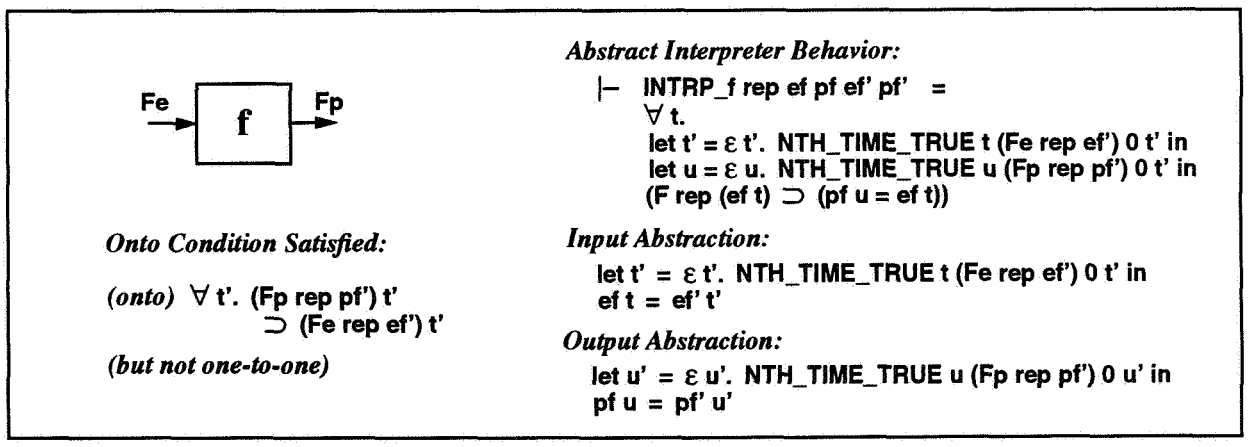


Figure 4.8: Example Surjective Multirate Interpreter Model.

The predicate **F rep** is assumed to evaluate to true precisely when the interpreter input is identified as one that causes an output event (over **pf**). In such a case the output, at time **u**, follows the input, at time **t**, in this example. This is the same behavior as for the injective model of Figure 4.4.

An interesting difference from before, however, is evident for the case when **F rep** is not true. In this scenario there is no output event over **pf**, therefore, output abstract time effectively stands still. Thus, for this scenario we cannot express the output as some sort of 'no-op' value as might be attempted for a normal state machine. This explains the need for the conditioning contained in the model.

We note here that this need to be able to underspecify the output behavior effectively rules out functional-behavior interpreter models for this application. Of the other interpreter models cited in Section 2, only Herbert's RTS approach provides the necessary modeling flexibility.

As in the last section, to provide a better appreciation for the practical importance of surjective modeling we present an example. Figure 4.9 shows it. As might be expected, instead of having the multiple subsystems on the input side of the system as before, here they are on the output side. As before, this system also mimics the PIU, with the subsystems **f'**, **g'**, and **h'** representing the P-Port, M-Port, and C-Port, respectively.

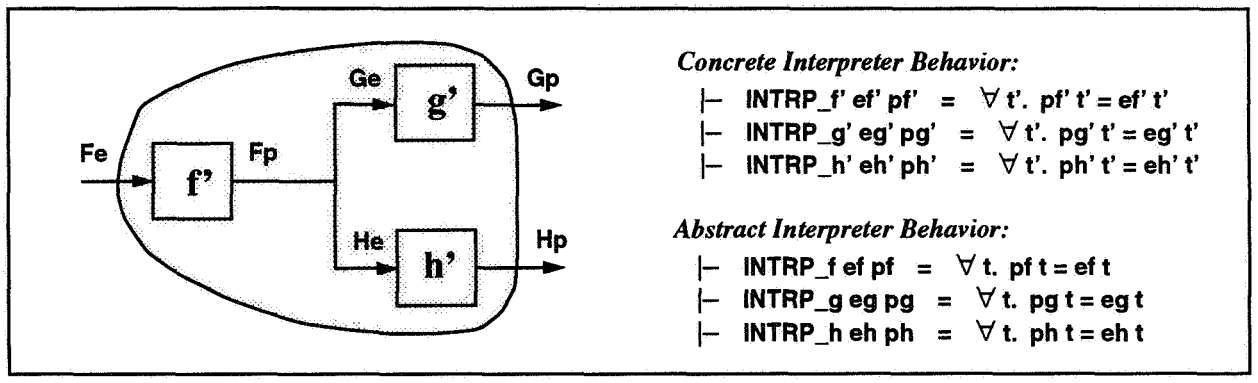


Figure 4.9: Example System Exhibiting Surjective Multirate Behavior.

In this example system, an output transaction arriving at input **ef** causes an output transaction at either output **pg** or else **ph**, but not both. In the context of the PIU, this represents L-Bus inputs being forwarded to the M-Bus or C-Bus, respectively. We are ignoring the PIU R-Port register file as a possible destination here.

The preferred system model for this configuration is shown in Figure 4.10. All signals are expressed using the same temporal operator, **t**, which counts input transactions. As above, the outputs are conditioned on the destination status contained within the system input. Although we have not proven this, we believe that this conditioning can be replaced here by a functional version. In other words, the **pg** expression could be written as: **pg t = (G rep (ef t)) ⇒ ef t | *'idle'***, where the ***'idle'*** output would represent an output tri-stated condition. This appears feasible, first of all, because the necessary temporal events exist (at the input), and secondly, because the output is presumed to be tristated anyway between the more coarse-grained output events. The overall benefit from doing this is not clear though.

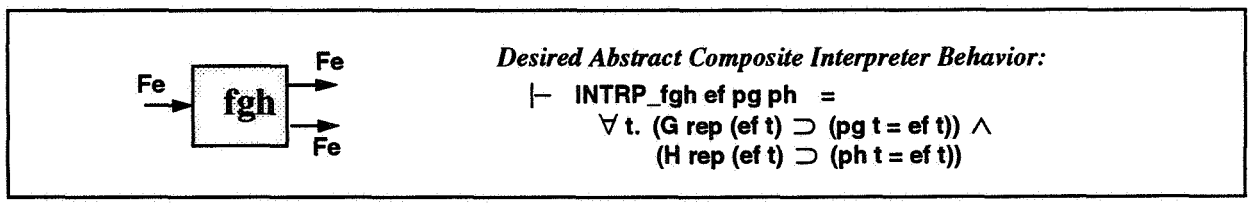


Figure 4.10: Desired Composite-System Behavior for System of Figure 4.9.

The specification hierarchy for the surjective cases looks remarkably similar to that for the injective case of the last section. We are currently in the process of completing the proofs for this case however. They will be made available when they are finished.

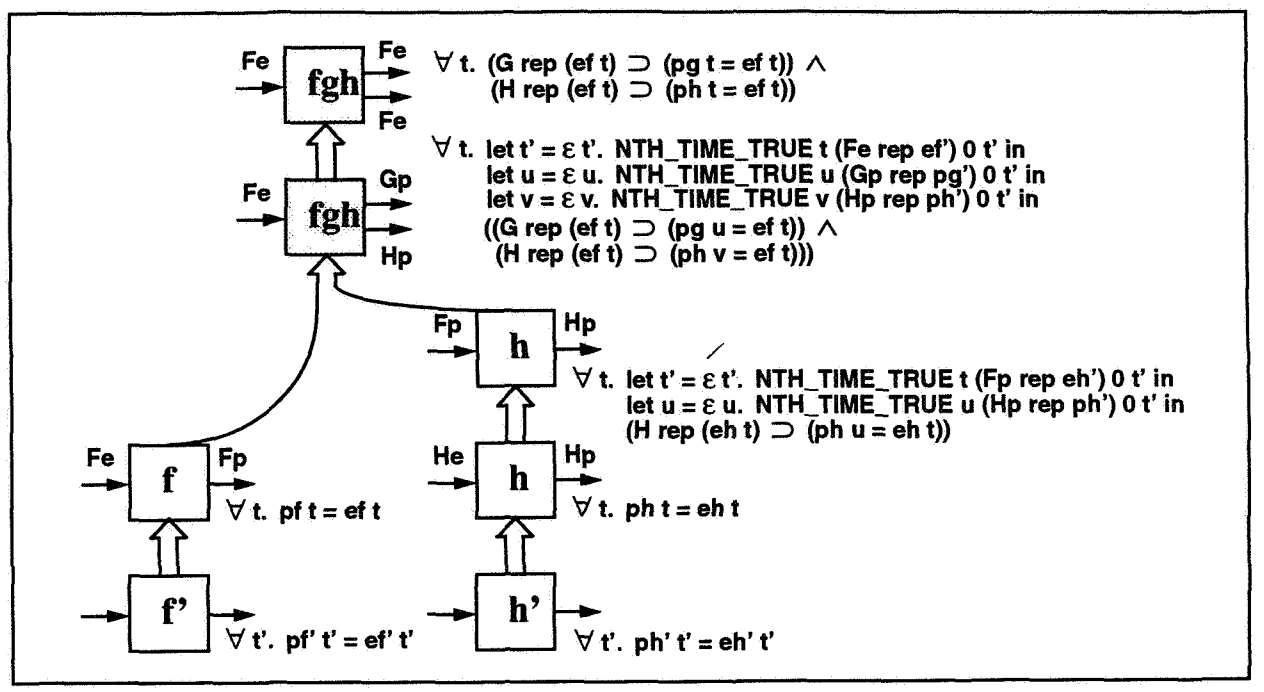


Figure 4.11: Partial Specification Hierarchy for System of Figure 4.9.

5 Processor-Memory Module Specification and Verification

In this section we overview our progress on the Processor-Memory Module (PMM) specification and verification. Section 5.1 overviews the PMM itself. The three sections that follow cover our work on the specification of the local processor, local memory, and PIU.

5.1 PMM Overview

The PMM is the primary processing subsystem of the Fault Tolerant Embedded Processor (FTEP) system. As seen in Figure 5.1, the PMM contains four major subsystems: the local processors, the local memory, the Processor Interface Unit (PIU), and the Core Bus (C_Bus) interface.

The PMM processors (CPU0 and CPU1) are arranged in a cold-sparing configuration to enhance long-life operation. Only one processor is active during a given mission. The choice of active processor is determined during initialization. The spare processor is disabled by the PIU through assertion of the processor's *cpu_reset* input. For the first implementation of the PMM, described in this report, Intel 80960MC microprocessors [Int89] are used for the local processors. They communicate with the PIU using the L_Bus bus protocol of the 80960.

Processor programs and data are stored in local electrically-erasable programmable read-only memory (EEPROM) and static random access memory (SRAM), respectively. Memory accesses are initiated by either the local processor or an external block acting as C_Bus master. In either case the PIU provides the memory interface. The features provided by the PIU include memory error correction, memory locking to implement atomic read-modify-write operations, byte accesses, and block accesses of up to 64 words. EEPROM and SRAM memory capacity in the first implementation is 1 MB (megabyte) of actual information storage each, implemented within seven 256Kx8-bit memory chips each. A (7,4) Hamming code provides single-bit error correction on memory reads.

The PIU also provides processor support features such as timers and interrupt control. Two 64-bit timers can be set by the processor to provide either timekeeping or watchdog functions. Processor interrupts are

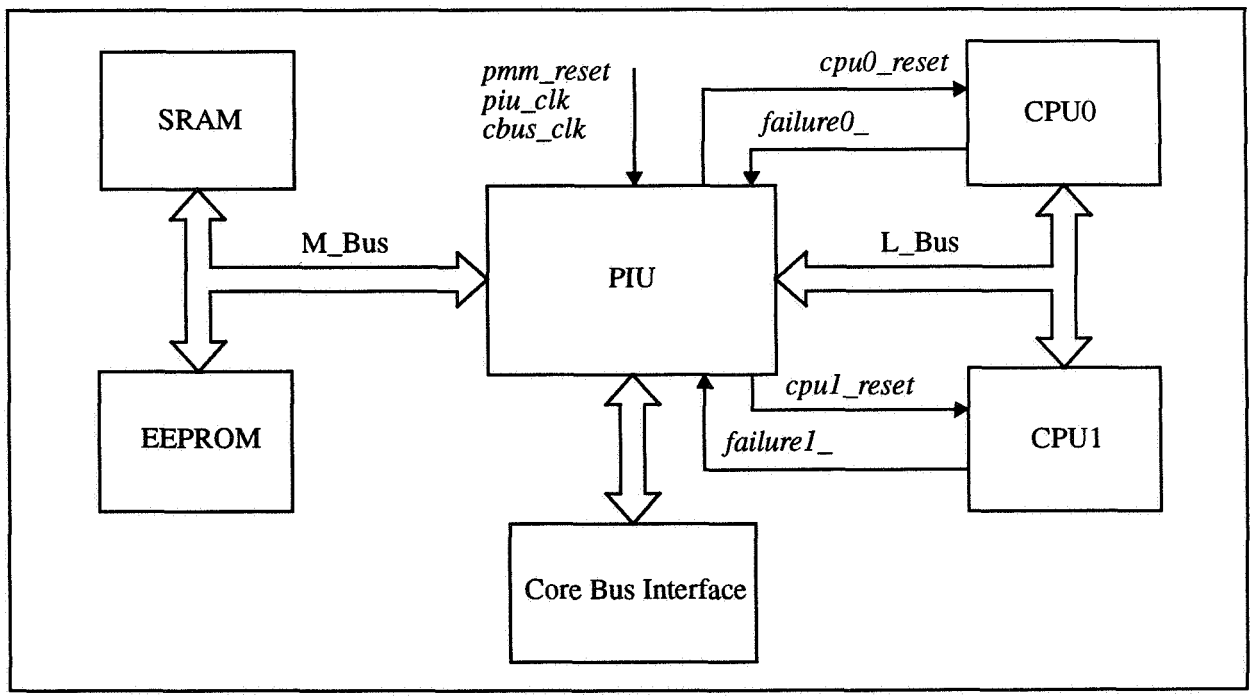


Figure 5.1: Block Diagram of the Processor-Memory Module.

generated within the PIU under two conditions. One condition is a timer time-out; the other is a write operation to a specially designated PIU register by either the local processor or C_Bus master.

The reset and clock signals shown at the top of Figure 5.1 are produced by the Fault-Tolerant Clock Unit (FTCU) not shown here. The *pmm_reset* signal is sent only to the PIU to allow it greater control over the local processors. For example, the PIU uses this signal to enter its initialization mode, during which it activates the processor reset signals. All of the PIU input signals produced by the FTCU are synchronized with those in the PIUs in redundant PMMs of a fault-tolerant FTEP core.

The structure of the PIU itself is shown in Figure 5.2. The Processor Port (P_Port), C_Bus Port (C_Port), and Memory Port (M_Port) implement the communication protocols for the L_Bus, C_Bus, and M_Bus, respectively. The M_Port also implements (7,4) Hamming encoding and decoding on writes and reads, respectively, to the local memory, and the C_Port implements single-bit parity encoding and decoding for C_Bus transfers.

The Register Port (R_Port) is the fourth, and final, port residing on the PIU's Internal Bus (I_Bus). It contains a state machine, counters, and various command and status registers used by the local processor to implement timers and interrupts.

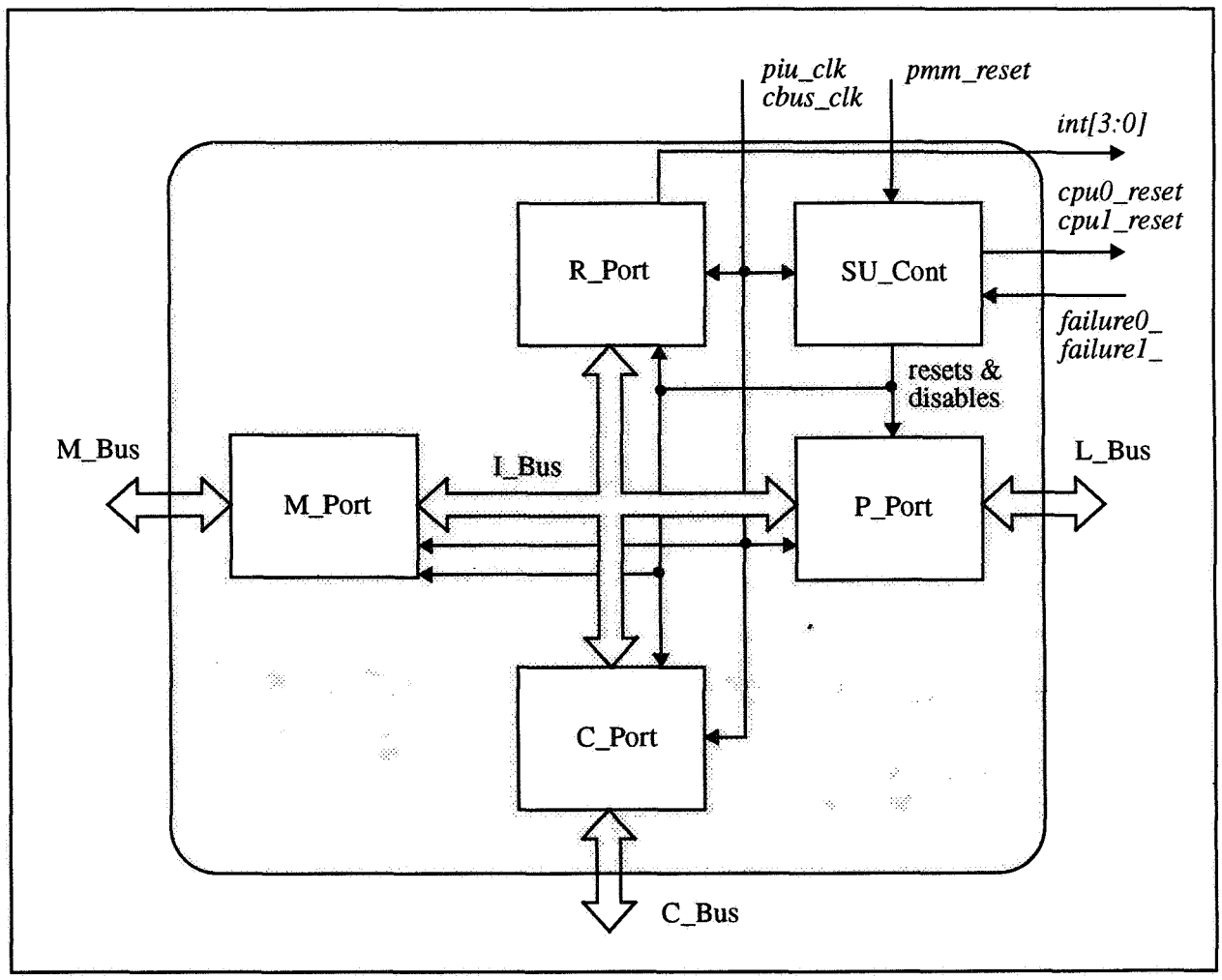


Figure 5.2: Major Blocks of the Processor Interface Unit.

The Start-up Controller (SU_Cont) implements the PMM initialization sequence. After it has concluded initialization, control is turned over to the other ports with the SU_Cont continuing operation in a background mode. The SU_Cont is not physically located on the I_Bus; however, for convenience, we will sometimes refer to it as one of the five PIU *ports*.

As explained in [Fur93a] the PMM exhibits at least four distinct types of behavior: (a) initialization and self-test, (b) timekeeping and interrupt control, (c) C_Bus-initiated memory requests to the PMM local memory and the PIU register file, and finally (d) local-processor-initiated memory requests to the same two destinations, plus to global memory via the C_Bus. For the purposes of this report, we are only concerned with the last type of behavior, local processor memory requests to the local memory, the internal PIU register file, and global memory.

5.2 Local Processor Specification

To support the Task 12 research focus on PMM composition, we have developed a simple transaction-level model for the local processor as a means to exercise the L_Bus interface to the PIU. The following HOL segment shows the transaction-level postcondition for this model. As evident from this definition, the processor model acts principally as a source and sink for L-Bus packets. The values for the transmitted packets are taken from the local state, **s**, and the data returned from the PIU on reads are simply written back to this state.

The packet opcode is determined by the address bits and read/write information. (The functions **SUBARRAY** and **ELEMENT** return a subarray and element of an array, respectively.) The six opcodes shown in the model are for reads and writes to the local memory (**_LM**), PIU register file (**_PIU**), and global memory (**_CB**). The address, block size, byte enables, and lock bit are transmitted directly. The output data is transmitted only on write operations, as expected, and the local state is updated only on reads. The Task 10 Specification Report [Fur93a] contains a more detailed explanation of these fields. The actual HOL listings for the local processor specification are contained in Appendix C.

Local Processor Postcondition:

```
|- CPUT_POSTC cputi s (er,el) p t  =
    let lmem = (¬ ELEMENT (CPUT_addrS (s t)) 31) ∧
               (¬ (SUBARRAY (CPUT_addrS (s t)) (25,24) = WORDN 1 3)) in
    let piu = (¬ ELEMENT (CPUT_addrS (s t)) 31) ∧
              (SUBARRAY (CPUT_addrS (s t)) (25,24) = WORDN 1 3) in
    let cbus = ELEMENT (CPUT_addrS (s t)) 31 in
    let write = CPUT_wrS (s t) in
    ((PBM_Opcode (p t) = lmem ⇒ (write ⇒ PBM_WrLM | PBM_RdLM) |
                          piu ⇒ (write ⇒ PBM_WrPIU | PBM_RdPIU) |
                       %cbus% (write ⇒ PBM_WrCB | PBM_RdCB))  ∧
     (PBM_Addr (p t) = CPUT_addrS (s t))  ∧
     (write ⊃ (PBM_Data (p t) = CPUT_dataS (s t)))  ∧
     (PBM_BS (p t) = CPUT_bsS (s t))  ∧
     (PBM_BE (p t) = CPUT_be_S (s t))  ∧
     (PBM_Lock (p t) = CPUT_lock_S (s t))  ∧
     (¬ write ⊃ (CPUT_dataS (s (t+1)) = PBS_Data (el t))))
```

5.3 Local Memory Specification

The transaction-level model for the local memory reflects the simple behavior of this subsystem. The following HOL code shows the postcondition for the local memory. It describes the expected behavior with the major point of interest being the handling of byte writes. Among the three M_Bus opcodes, **MBM_Wr** represents word writes, **MBM_Rd** represents reads, and **MBM_RdWr** represents byte writes. As indicated in this definition, an opcode of **MBM_RdWr** corresponds to a 'simultaneous' read and write. Actually, at the implementing clock level the read precedes the write. The PIU executes a read-modify-write operation to implement byte writes so that the full 32-bit memory word gets reencoded.

The function **MALTER** in this definition inserts into its first argument (the memory) the value of the third argument (the M_Bus data array). The insertion is done within the address bounds of the second argument. If the operation is either a word write or a byte write then the memory is updated with the new value, otherwise the old value is used. The opcode output of **MBS_Ready** indicates that the memory is implementing its part of the M_Bus protocol correctly. For the simple memory-bus protocols of typical memory chips, this means that the memory is not driving the concrete-level data signal lines out of turn. On data reads and byte writes the memory system returns the requested data array.

Local Memory Postcondition:

```
|- MEMT_POSTC memti s e p t  =
   let adr_min = VAL 23 (MBM_Addr (e t)) in
   let bs = VAL 1 (MBM_BS (e t)) in
   let adr_max = adr_min + bs in
   ((MemtS (s (t+1)) =
       ((MBM_Opcode (e t) = MBM_Wr) ∨ (MBM_Opcode (e t) = MBM_RdWr))
         ⇒ MALTER (MemtS (s t))
                  (adr_max,adr_min)
                  (MBM_Data (e t))
           | MemtS (s t))  ∧
    (MBS_Opcode (p t) = MBS_Ready)  ∧
    (((MBM_Opcode (e t) = MBM_Rd) ∨ (MBM_Opcode (e t) = MBM_RdWr))
       ⊃ (MBS_Data (p t) = SUBARRAY (MemtS (s t)) (adr_max,adr_min))))
```

5.4 PIU Specification Refinements

During this task we have made some modifications to the existing PIU models, some by choice and others by necessity. In some cases we found that models could be greatly simplified by making better use of specially-defined functions. The M-Port model described in this section is a good example of such a case. Other simplifications were made by moving away from the functional modeling style of the GIT, and instead using a relational style.

Changes that were dictated to us primarily concern composition. In particular, it became necessary to redefine our data structures that had been defined with respect to the ports during Task 10. For example, the P-Port specification previously had its own set of state, input, and output structures that were different from those of the other ports. Distributing these signals among the other ports would have required additional processing that would have added even more complexity to an already sizeable PIU specification.

To correct this, in this task we have introduced new data structures for the buses, as well as the other interconnects of the PMM. These provide the expected improvement in sharing among the port models that use them. We have included the transaction-level data structures within the HOL listings in Appendix C.

In the rest of this section we describe one of the revised port models, namely the transaction-level post-condition for the M-Port. Much of this definition is familiar from the earlier models. The interesting parts here concern the Hamming encoding and decoding, plus the implementation of byte writes.

Beginning with the simpler Hamming decoding on memory reads, the I-Bus output data is shown here being processed by a function **MAPN**. This function creates an array of size **bs** by mapping **(Ham_Dec rep)** over the individual elements of the M-Bus data input.

Modeling byte-write operations is inherently complex in a word-addressed memory, especially when the data are encoded. For each word to be written, the M-Port first reads the current word from memory, then inserts the new bytes into the word (based on the byte enable bit pattern), and finally writes the updated word back into memory. The word read from memory is decoded before being used and the updated word is encoded before being stored.

In the M-Port postcondition, the function **MAP_LISTN** creates an array of size **bs** similar to **MAPN** above. However, rather than applying a single function over the entire input array, it applies a list of functions. The first function of the list is applied to the first element of the array, and so on. The function **BYTE_MUXN** performs the desired byte selection here for each word when mapped over its four arguments: **31**, **be**, the 'new' word, and the 'old' word.

M-Port Postcondition:

```
⊢ MT_POSTC rep mti (ei,em,er) (pm,pi) t =
  let bs = VAL 1 (PBM_BS (ei t)) in
  ((IBS_Opcode (pi t) = IBS_Ready) ∧
   ((mti = MT_Rd)
     ⊃ (MBS_Opcode (em t) = MBS_Ready)
         ⊃ (IBS_Data (pi t) = MAPN bs (Ham_Dec rep) (MBS_Data (em t)))) ∧
   (MBM_Opcode (pm t) =
      (mti = MT_Rd) ⇒ MBM_Rd |
      ((mti = MT_Wr) ∧ (PBM_BE (ei t) = WORDN 3 15)) ⇒ MBM_Wr |
      MBM_RdWr) ∧
   (MBM_Addr (pm t) = PBM_Addr (ei t)) ∧
   ((mti = MT_Wr)
     ⊃ (MBM_Data (pm t) =
          let be = PBM_BE (ei t) in
          MAP_LISTN bs
            [(Ham_Enc rep) o
             (λf. BYTE_MUXN 31 be (ELEMENT (PBM_Data (ei t)) 0) f);
             (Ham_Enc rep) o
             (λf. BYTE_MUXN 31 be (ELEMENT (PBM_Data (ei t)) 1) f);
             (Ham_Enc rep) o
             (λf. BYTE_MUXN 31 be (ELEMENT (PBM_Data (ei t)) 2) f);
             (Ham_Enc rep) o
             (λf. BYTE_MUXN 31 be (ELEMENT (PBM_Data (ei t)) 3) f)]
            (MAPN bs (Ham_Dec rep) (MBS_Data (em t))))) ∧
   (MBM_BS (pm t) = PBM_BS (ei t)))))
```

Compared to the old style of specification that we described in our Task 10 reports, the new models are much more compact, and arguably easier to understand. The difficulty in comprehension that is introduced by the use of the new functions is more than offset by the reduction in bulk. Nevertheless, we believe that specifications of this sort should be challenged either through proof or by simulation. In the future we hope to address the issue of requirements specification correctness with additional rigor.

6 Conclusions

The ability to compose hardware interpreter models at high levels of abstraction is fundamental to the practical specification and formal verification of nontrivial hardware systems. Without such a capability, large-scale system verifications would quickly become bogged down by the explosion in low-level state resulting from the independent actions of typical interacting subsystems. In the work reported here, we have developed a technique for high-level interpreter composition and demonstrated its efficacy on a substantial verification example. The approach is suitable for application to the transaction-level verification of the target Processor-Memory Module (PMM).

To our knowledge, all formal hardware verifications to date have performed their component compositions at very low levels of abstraction, comparable to the PMM clock level. At these levels of abstraction, the presence of a global clock greatly facilitates the necessary composition theorem proving. The global clock is the basis for a single temporal abstraction definition that is shared by each component interface. This common abstraction is, in turn, instrumental for achieving a straightforward composition proof.

The higher levels of abstraction have no such notions of a global clock. Instead, at these levels subsystems communicate by using what are normally termed *protocols*. These protocols are defined with respect to certain events defined over the low-level signals that interconnect the communicating subsystems. A formal proof of high-level composition must therefore focus on these shared events as a means to synchronize subsystem interactions.

In this 'interface-event clocking' approach, the abstract-level temporal operators for a subsystem's input and output signals measure the accumulated count of the appropriate transaction 'events' occurring over the input and output interfaces, respectively. This view of temporal abstraction introduces some interesting modeling and verification issues. Because the events for a subsystem's inputs and outputs are defined for different interfaces, they may or may not occur in lock step. When they don't, then the input and output events count at different rates. This has the effect of creating interpreter behavior functions where the time variables for the inputs and outputs are different.

Such 'multiclocked interpreters' have great practical importance in hardware modeling and verification. For example, within the PMM there are scenarios in which the relationship between interpreter input events and output events is one-to-one, but not onto, and vice versa. In other words, there are cases in which a subsystem's input events cause only a subset of its output events, and cases where only a subset of the input events cause output events. In fact, these scenarios represent an important class of computing hardware structures that includes, for example, bus-based shared memory systems.

We have developed techniques that allow us to create and formally verify such 'multiclocked interpreters' with respect to standard interpreter models. We have also implemented an approach to formally verify compositions involving multiclocked interpreters. Finally, we have formally verified standard interpreters with respect to multiclocked interpreter models. These techniques were all demonstrated on a substantial example verification problem.

In addressing the composition needs of a real-world subsystem, we have exceeded the capabilities of interpreter modeling approaches developed for streamlined microprocessor verifications. Prior work under Task 10 showed inadequacies in their handling of abstraction. In this task, we have found that multirate interpreters cannot be modeled using the functional style of behavior embedded in some standard models. A relational style of behavior modeling is required here.

The hierarchical pre-post logic (HPL) modeling approach that was developed largely under this contract has been shown to satisfy the modeling and verification needs of the PMM. Future work should focus on exploring further the basic problems in PMM verification that will surely arise as this effort continues under Boeing funding. Beyond this, we should consider embedding HPL's multiclocked interpreter approach to composition within a generic theory, a composition-oriented version of Windley's generic interpreter theory.

7 References

[Chu40] A. Church, "A Formulation of the Simple Theory of Types," *Journal of Symbolic Logic*, Vol. 5, 1940.

[Coh88] A. Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level," in G. Birtwistle and P.A. Subrahmanyam (eds.), *VLSI Specification, Verification, and Synthesis*, Kluwer Academic Publishers, 1988, pp. 27–71.

[Coh89] A. Cohn, "Correctness Properties of the Viper Block Model. The Second Level," in G. Birtwistle and P.A. Subrahmanyam (eds.), *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989, pp. 1–91.

[Con86] R.L. Constable, *Implementing Mathematics with the NUPRL Proof Development System*, Prentice Hall, 1986.

[Fur93a] D.A. Fura, P.J. Windley, and G.C. Cohen, "Towards the Formal Specification of the Requirements and Design of a Processor Interface Unit," *NASA Contractor Report 4521*, 1993.

Fur93b] D.A. Fura, P.J. Windley, and G.C. Cohen, "Towards the Formal Verification of the Requirements and Design of a Processor Interface Unit," *NASA Contractor Report 4522*, 1993.

[Fur94] D.A. Fura, *Formal Abstraction, Composition, and Verification Methods for Fault-Tolerant Hardware Interpreters*, Ph.D. thesis, Department of Electrical Engineering, University of Washington, 1994.

[Gor79] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[Gor93] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[Gra92] B.T. Graham, *The SECD Microprocessor: A Verification Case Study*, Kluwer Academic Publishers, 1992.

[Her88] J. Herbert, "Temporal Abstraction of Digital Designs," Technical Report No. 122, Computer Laboratory, University of Cambridge, February 1988.

[Her92] J.M.J. Herbert, "Incremental Design and Formal Verification of Microcoded Microprocessors," in V. Stavridou, T.F. Melham, and R.T. Boute (eds.), *Theorem Provers in Circuit Design*, Elsevier Science Publishers, 1992, pp. 157–174.

[Hun86] W.A. Hunt, Jr., *FM8501: A Verified Microprocessor*, Ph.D. thesis and Technical Report 47, Institute for Computing Science, University of Texas at Austin, February 1986.

[Int89] Intel Corporation, *80960MC Hardware Designer's Reference Manual*, June 1989.

[Joy90] J.J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems*, Ph.D. thesis and Technical Report No. 195, Computer Laboratory, University of Cambridge, May 1990.

[Kan87] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, 1987.

[Lev93] K. Levitt, et. al., "Formal Verification of a Microcoded VIPER Microprocessor Using HOL," *NASA Contractor Report 4489*, February 1993.

[McM93] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[Mel90] T. Melham, *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logi*, Ph.D. thesis and Technical Report No. 201, Computer Laboratory, University of Cambridge, August 1990.

[Win90] P.J. Windley, *The Formal Verification of Generic Interpreters*, Ph.D. thesis and Research Report CSE-90-22, Division of Computer Science, University of California, Davis, July 1990.

[Win91] P. Windley, K. Levitt, and G.C. Cohen, "The Formal Verification of Generic Interpreters," *NASA Contractor Report 4403*, October 1991.

[Win92] P.J. Windley, "Abstract Theories in HOL," in *Proceedings of the 1992 International Conference on the HOL theorem Prover and its Application*, October 1992.

Appendix A: HOL Overview

HOL is a general theorem proving system developed at the University of Cambridge [Gor93] that is based on Church's theory of simple types, or higher order logic [Chu40]. Church developed higher order logic as a foundation for mathematics, but it can be used for describing and reasoning about computational systems of all kinds. Higher order logic is similar to the more familiar predicate logic, but allows quantification over predicates and functions, not just variables, allowing more general systems to be described.

HOL grew out of Robin Milner's LCF theorem prover [Gor79] and is similar to other LCF progeny such as NUPRL [Con86]. Because HOL is the theorem proving environment used in the body of this work, we describe it in more detail. This description is taken from [Win90].

HOL's proof style can be tailored to the individual user, but most users find it convenient to work in a goal-directed fashion. HOL is a tactic-based theorem prover. A tactic breaks a goal into one or more subgoals and provides a justification for the goal reduction in the form of an inference rule. Tactics perform tasks such as induction, rewriting, and case analysis. At the same time, HOL allows forward inference, and many proofs are a combination of forward and backward proof styles. Any theorem-proving strategy a user employs in connection with HOL is checked for soundness, eliminating the possibility of incorrect proofs.

HOL provides a metalanguage, ML, for programming and extending the theorem prover. Using ML, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be combined into new theories for later use. The metalanguage makes the HOL verification system extremely flexible.

In HOL, all proofs, even tactic-based proofs, are eventually reduced to the application of inference rules. Most nontrivial proofs require large numbers of inferences. Proofs of large devices such as microprocessors can take many millions of inference steps. In a proof containing millions of steps, what kind of confidence do we have that the proof is correct? One of the most important features of HOL is that it is *secure*, meaning that new theorems can only be created in a controlled manner. HOL is based on five primitive axioms and eight primitive inference rules. All high-level inference rules and tactics do their work through some combination of the primitive inference rules. Because the entire proof can be reduced to one using only eight primitive inference rules and five primitive axioms, an independent proof-checking program could check the proof syntactically.

A.1 The Language

The object language of HOL is described in this section. We will discuss HOL's terms and types.

Terms. All HOL expressions are made up of terms. There are four kinds of terms in HOL: variables, constants, function applications, and abstractions (lambda expressions). Variables and constants are denoted by any sequence of letters, digits, underlines, and primes starting with a letter. Constants are distinguished in the logic; any identifier that is not a distinguished constant is taken to be a variable. Constants and variables can have any finite arity, not just 0, and, thus, can represent functions as well.

Function application is denoted by juxtaposition, resulting in a prefix syntax. Thus, a term of the form "**t1 t2**" is an application of the operator **t1** to the operand **t2**. The term's value is the result of applying **t1** to **t2**.

An abstraction denotes a function and has the form "λ **x. t**." An abstraction "λ **x. t**" has two parts: the bound variable **x** and the body of the abstraction **t**. It represents a function, **f**, such that "**f(x) = t**." For example, "λ **y. 2*y**" denotes a function on numbers that doubles its argument.

Constants can belong to two special syntactic classes. Constants of arity 2 can be declared to be infix. Infix operators are written: "**rand1 op rand2**" instead of in the usual prefix form: "**op rand1 rand2.**" Table A.1 shows several of HOL's built-in infix operators.

Constants can also belong to another special class called binders. A familiar example of a binder is ∀ If **c** is a binder, then the term "**c x. t**" (where **x** is a variable) is written as shorthand for the term "**c(λ x. t)**." Table A.2 shows several of HOL's built-in binders.

Table A.1: HOL Infix Operators.

Operator	*Application*	*Meaning*
=	**t1 = t2**	**t1** equals **t2**
,	**t1, t2**	the pair **t1** and **t2**
∧	**t1 ∧ t2**	**t1** and **t2**
∨	**t1 ∨ t2**	**t1** or **t2**
⊃	**t1 ⊃ t2**	**t1** implies **t2**

Table A.2: HOL Binders.

Binder	*Application*	*Meaning*
∀	**∀ x. t**	for all **x**, **t**
∃	**∃ x. t**	there exists an **x** such that **t**
ε	**ε x. t**	choose an **x** such that **t** is true

In addition to the infix constants and binders, HOL has a conditional statement that is written "**a ⇒ b | c**," meaning "if **a** then **b** else **c**."

Types. HOL is strongly typed to avoid Russell's paradox and others like it. Russell's paradox occurs in a high order logic when one can define a predicate that leads to a contradiction. Specifically, suppose that we define **P** as **P(x) = ¬x(x)**, where ¬ denotes negation. **P** is true when its argument applied to itself is false. Applying **P** to itself leads to a contradiction since **P(P) = ¬P(P)** (i.e., true = false). This kind of paradox can be prevented by typing since, in a typed system, the type of **P** would never allow it to be applied to itself.

Every term in HOL is typed according to the following recursive rules:

a. Each constant or variable has a fixed type.
b. If **x** has type α and **t** has type β, the abstraction **λ x. t** has the type (α → β).
c. If **t** has the type (α → β) and **u** has the type α, the application **t u** has the type β.

Types in HOL are built from type variables and type operators. Type variables are denoted by a sequence of asterisks (*) followed by a (possibly empty) sequence of letters and digits. Thus, *, ***, and ***ab2** are all valid type variables. All type variables are universally quantified implicitly, yielding type polymorphic expressions.

Type operators construct new types from existing types. Each type operator has a name (denoted by a sequence of letters and digits beginning with a letter) and an arity. If $\alpha_1, \ldots, \alpha_n$ are types and **op** is a type operator of arity n, then $(\alpha_1, \ldots, \alpha_n)$ **op** is a type. Note that type operators are postfix while normal function application is prefix or infix. A type operator of arity 0 is a type constant.

HOL has several built-in types that are listed in Table A.3. The type operators **bool**, **ind**, and **fun** are primitive. HOL has a special syntax that allows (*,**)**prod** to be written as (* # **), (*,**)**sum** to be written as (* + **), and (*,**)**fun** to be written as (* → **).

Table A.3: HOL Type Operators.

Operator	*Arity*	*Meaning*
bool	0	booleans
ind	0	individuals
num	0	natural numbers
(*)list	1	lists of type *
(*,)prod**	2	products of * and **
(*,)sum**	2	coproducts of * and **
(*,)fun**	2	functions from * to **

A.2 The Proof System

HOL is not an automated theorem prover, but is more than simply a proof checker, falling somwhere between these two extremes. HOL has several features that contribute to its use as a verification environment:

a. Several built-in theories, including booleans, individuals, numbers, products, sums, lists, and trees. These theories contain the five axioms that form the basis of higher order logic, as well as a large number of theorems that follow from them.
b. Rules of inference for higher order logic. These rules contain not only the eight basic rules of inference from higher order logic, but also a large body of *derived* inference rules that allow proofs to proceed using larger steps. The HOL system has rules that implement the standard introduction and elimination rules for Predicate Calculus as well as specialized rules for rewriting terms.
c. A collection of tactics. Examples of tactics include: **REWRITE_TAC** which rewrites a goal according to some previously proven theorem or definition; **GEN_TAC** which removes unnecessary universally quantified variables from the front of terms; and **EQ_TAC** which says that to show two things are equivalent, we should show that they imply each other.
d. A proof management system that keeps track of the state of an interactive proof session.
e. A metalanguage, ML, for programming and extending the theorem prover. Using the metalanguage, tactics can be put together to form more powerful tactics, new tactics can be written, and theorems can be aggregated to form theories for later use. The metalanguage makes the verification system extremely flexible.

Appendix B: Interpreter Verification Test Case Studies

This section contains HOL theories for several interpreter test case studies. The theory intrp contains the justification theorem for standard interpreters. The theory *bi_intrp* contains the justification theorem for monorate bijective interpreters. The theory *in_intrp* contains theorems demonstrating the feasibility of formal composition for multirate injective interpreters.

B.1 Standard Interpreter Justification-Proof Example

```
%----------------------------------------------------------------------------

   File:     intrp.ml

   Author:   (c) D.A. Fura 1993

   Date:     30 September 1993

   Converts a set of three proof obligations (INTRP_PREC, PREC_SAT, and SEQ_LIVE)
   into result INTRP.

----------------------------------------------------------------------------%

set_flag ('timing',true);;

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/ftep/piu/hol/pport/pproc/';
                                  '/home/elvis6/dfura/hol/ml/';
                                  '/home/elvis6/dfura/hol/Library/tools/';
                                  '/home/elvis6/dfura/hol/Library/abs_theory/'
                                 ]);;

system 'rm intrp.th';;

new_theory 'intrp';;

loadf 'abstract';;
loadf 'aux_defs';;

map load_parent ['ineq';'templogic_def';'assoc'];;

new_type_abbrev ('time', ":num");;
new_type_abbrev ('time'', ":num");;

load_library 'reduce';;

loadf 'pt_tacs.ml';;

let REP_lemma = new_abstract_representation
   [('EXEC', ":(*instr->(time->*env)->(time->*out)->time->bool)");
    ('PREC', ":(*instr->(time->*env)->(time->*out)->time->bool)");
    ('POSTC', ":(*instr->(time->*env)->(time->*out)->time->bool)")
   ];;

make_inst_thms REP_lemma;;

let REP = abstract_type 'intrp' 'EXEC';;

%----------------------------------------------------------------------------
  Definitions for interpreter.
----------------------------------------------------------------------------%

let INTRP = new_definition
   ('INTRP',
    "! (rep :^REP) (e :time->*env) (p :time->*out) .
     INTRP rep e p =
        !k t. EXEC rep k e p t
              ==> POSTC rep k e p t"
   );;
```

```
let INTRP_PREC = new_definition
   ('INTRP_PREC',
    "! (rep :^REP) (e :time->*env) (p :time->*out) .
     INTRP_PREC rep e p =
        !k t. EXEC rep k e p t /\
              PREC rep k e p t
              ==> POSTC rep k e p t"
   );;

let PREC_SAT = new_definition
   ('PREC_SAT',
    "! (rep :^REP) (e :time->*env) (p :time->*out) .
     PREC_SAT rep e p =
        (!k. EXEC rep k e p 0 ==> PREC rep k e p 0) /\
        (!k k1 t. POSTC rep k e p t /\
                  EXEC rep k1 e p (SUC t)
                  ==> PREC rep k1 e p (SUC t))"
   );;

let SEQ_LIVE = new_definition
   ('SEQ_LIVE',
    "! (rep :^REP) (e :time->*env) (p :time->*out) .
     SEQ_LIVE rep e p =
        !k t. EXEC rep k e p (SUC t)
              ==> ?k1. EXEC rep k1 e p t"
   );;

let FILTER_RULE_ASSUM_TAC filter rule :tactic =
   let frule = (\thm. (filter (concl thm)) => (rule thm) | thm) in
   POP_ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (rev (map frule asl)));;

let PRIOR_EXEC' = TAC_PROOF
   (([],
     "! (rep :^REP) (e :time->*env) (p :time->*out) .
      SEQ_LIVE rep e p ==>
       (!u t1 k. EXEC rep k e p (t1+u) ==>
                 (t1 < (t1+u)) ==>
                  (?k1. EXEC rep k1 e p t1))"),
    REWRITE_TAC [SEQ_LIVE]
    THEN REPEAT GEN_TAC
    THEN DISCH_TAC
    THEN INDUCT_TAC
    THEN REWRITE_TAC [ADD_CLAUSES;LESS_REFL]
    THEN REPEAT STRIP_TAC
    THEN RES_TAC
    THEN ASM_CASES_TAC "u = 0"
    THENL [
      POP_ASSUM (\thm. RULE_ASSUM_TAC (REWRITE_RULE [thm;ADD_CLAUSES]))
      THEN EXISTS_TAC "k1:*instr"
    ;
      IMP_RES_TAC LESS_ADD_NONZERO
      THEN RES_TAC
      THEN FILTER_RULE_ASSUM_TAC
             (\tm. tm = "!m. m < (m + u)")
             (\thm. SPEC "t1:time" thm)
      THEN RES_TAC
      THEN EXISTS_TAC "k1''':*instr"
    ]
    THEN ASM_REWRITE_TAC[]
   );;

let PRIOR_EXEC = TAC_PROOF
   (([],
     "! (rep :^REP) (e :time->*env) (p :time->*out) .
      SEQ_LIVE rep e p ==>
       (!t t1 k. EXEC rep k e p t ==>
                 (t1 < t) ==>
                  (?k1. EXEC rep k1 e p t1))"),
    REPEAT GEN_TAC
    THEN DISCH_TAC
    THEN IMP_RES_TAC PRIOR_EXEC'
```

```
    THEN REPEAT STRIP_TAC
    THEN FILTER_RULE_ASSUM_TAC
           (\tm. is_forall tm)
           (\thm. (SPECL ["k:*instr";"t1:time";"t-t1"] thm))
    THEN ASSUME_TAC (SPEC "t1:time" LESS_EQ_REFL)
    THEN IMP_RES_TAC LT_IMP_LE
    THEN IMP_RES_TAC
           (SPECL ["t1:time";"t1:time";"t:time"] (SYM_RULE ASSOC_SUB_ADD1))
    THEN POP_ASSUM (\thm. ALL_TAC)
    THEN POP_ASSUM (\thm. RULE_ASSUM_TAC
                            (REWRITE_RULE [thm;SUB_EQUAL_0;ADD_CLAUSES]))
    THEN RES_TAC
    THEN EXISTS_TAC "k1:*instr"
    THEN ASM_REWRITE_TAC[]
   );;

let EXEC_IMP_PREC = TAC_PROOF
   (([],
     "! (rep :^REP) (e :time->*env) (p :time->*out) .
      INTRP_PREC rep e p ==>
        PREC_SAT rep e p ==>
          SEQ_LIVE rep e p ==>
            (!t k. EXEC rep k e p t ==> PREC rep k e p t)"),
    REWRITE_TAC [INTRP_PREC;PREC_SAT]
    THEN REPEAT GEN_TAC
    THEN STRIP_TAC
    THEN STRIP_TAC
    THEN STRIP_TAC
    THEN INDUCT_TAC
    THEN ASM_REWRITE_TAC[]
    THEN ASSUME_TAC (REWRITE_RULE [LE_EQ_LT_SUC] (SPEC "t:time" LESS_EQ_REFL))
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC PRIOR_EXEC
    THEN RES_TAC
    THEN RES_TAC
   );;

let INTRP_CORR = TAC_PROOF
   (([],
     "! (rep :^REP) (e :time->*env) (p :time->*out) .
      INTRP_PREC rep e p ==>
        PREC_SAT rep e p ==>
          SEQ_LIVE rep e p ==>
            INTRP rep e p"),
    REWRITE_TAC [INTRP]
    THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC EXEC_IMP_PREC
    THEN RULE_ASSUM_TAC (REWRITE_RULE [INTRP_PREC])
    THEN RES_TAC
   );;

close_theory();;
```

B.2 Monorate Bijective Interpreter Justification-Proof Example

```
%----------------------------------------------------------------------------

   Filename:    bi_intrp.ml

   Author:      (c) D.A. Fura 1993

   Date:        30 September 1993

   Converts a set of four proof obligations (TRANS_ONE_TO_ONE, TRANS_ONTO,
   FIRST_TRANS_CAUSAL, and OUT_TRANS_LIVE) into result NEXT_OUT_TRANS_IS_NTH.

----------------------------------------------------------------------------%

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/hol/ml/';
```

```
                                '/home/elvis6/dfura/hol/Library/tools/'
                               ]);;

set_flag ('timing',true);;

system 'rm bi_intrp.th';;

new_theory 'bi_intrp';;

loadf 'aux_defs';;

map load_parent ['ineq';'templogic_def'];;

%----------------------------------------------------------------------
   Proof obligations.
----------------------------------------------------------------------%

let TRANS_ONE_TO_ONE = new_definition
   ('TRANS_ONE_TO_ONE',
    "TRANS_ONE_TO_ONE Ee Ep =
       ! t te' tp'.
       NTH_TIME_TRUE t Ee 0 te' ==>
        STABLE_FALSE_THEN_TRUE Ep (te',tp') ==>
         TRUE_THEN_STABLE_FALSE Ee (te',tp')"
   );;

let TRANS_ONTO = new_definition
   ('TRANS_ONTO',
    "TRANS_ONTO Ee Ep =
       ! t te' tp''.
       NTH_TIME_TRUE t Ep 0 tp'' ==>
        STABLE_FALSE_THEN_TRUE Ee (tp''+1,te') ==>
         TRUE_THEN_STABLE_FALSE Ep (tp'',te'-1)"
   );;

let FIRST_TRANS_CAUSAL = new_definition
   ('FIRST_TRANS_CAUSAL',
    "FIRST_TRANS_CAUSAL Ee Ep =
       ! te'.
       STABLE_FALSE Ee (0,te'-1) ==>
        STABLE_FALSE Ep (0,te'-1)"
   );;

let OUT_TRANS_LIVE = new_definition
   ('OUT_TRANS_LIVE',
    "OUT_TRANS_LIVE Ee Ep =
     ! t te'.
     NTH_TIME_TRUE t Ee 0 te' ==>
      ?tp'. STABLE_FALSE_THEN_TRUE Ep (te',tp')"
   );;

%----------------------------------------------------------------------
   Proof goal.
----------------------------------------------------------------------%

let NEXT_OUT_TRANS_IS_NTH = new_definition
   ('NEXT_OUT_TRANS_IS_NTH',
    "NEXT_OUT_TRANS_IS_NTH Ee Ep =
       ! t te' tp'.
       NTH_TIME_TRUE t Ee 0 te' ==>
        STABLE_FALSE_THEN_TRUE Ep (te',tp') ==>
         NTH_TIME_TRUE t Ep 0 tp'"
   );;

%----------------------------------------------------------------------
   Justification theorem.
----------------------------------------------------------------------%

let FILTER_RULE_ASSUM_TAC filter rule :tactic =
   let frule = (\thm. (filter (concl thm)) => (rule thm) | thm) in
   POP_ASSUM_LIST (\asl. MAP_EVERY ASSUME_TAC (rev (map frule asl)));;
```

```
let FILTER_POP_ASSUM filter ttac :tactic =
   \(asl,w).
   let thm,thml = remove filter asl in
   ttac (ASSUME thm) (thml,w);;

let LESS_EQ_STABLE_FALSE_THEN_TRUE = prove_thm
   ('LESS_EQ_STABLE_FALSE_THEN_TRUE',
    "! f t1 t2. STABLE_FALSE_THEN_TRUE f (t1,t2) ==> t1 <= t2",
     REWRITE_TAC [STABLE_FALSE_THEN_TRUE]
     THEN REPEAT STRIP_TAC
     THEN ASM_REWRITE_TAC[]
   );;

let JUST_THEOREM = TAC_PROOF
   (([],
     "! Ee Ep.
      TRANS_ONE_TO_ONE Ee Ep /\
      TRANS_ONTO Ee Ep /\
      FIRST_TRANS_CAUSAL Ee Ep /\
      OUT_TRANS_LIVE Ee Ep
      ==> NEXT_OUT_TRANS_IS_NTH Ee Ep"),
    REWRITE_TAC
      [TRANS_ONE_TO_ONE;TRANS_ONTO;FIRST_TRANS_CAUSAL;NEXT_OUT_TRANS_IS_NTH;
       OUT_TRANS_LIVE]
    THEN REPEAT GEN_TAC
    THEN REPEAT DISCH_TAC
    THEN INDUCT_TAC
    THENL [
      % Subgoal 1: base case %
      REWRITE_TAC [NTH_TIME_TRUE]
      THEN REPEAT GEN_TAC
      THEN ASM_CASES_TAC "te' = 0"
      THEN ASM_REWRITE_TAC[]
      THEN REPEAT STRIP_TAC
      THEN IMP_RES_TAC NOT_EQ_ZERO
      THEN IMP_RES_TAC GREATER
      THEN IMP_RES_TAC STABLE_FALSE_THEN
      THEN POP_ASSUM_LIST (MAP_EVERY (\thm. STRIP_ASSUME_TAC thm))
      THEN RES_TAC
      THEN IMP_RES_TAC SUP_INTERVAL_STABLE_FALSE_THEN_TRUE
      THEN IMP_RES_TAC (REWRITE_RULE [ADD1] LT_IMP_SUC_LE)
      THEN RULE_ASSUM_TAC (REWRITE_RULE[ADD_CLAUSES])
      THEN IMP_RES_TAC SUB_ADD
      THEN POP_ASSUM
             (\thm. RULE_ASSUM_TAC (REWRITE_RULE[thm;ZERO_LESS_EQ;LESS_EQ_REFL]))
      THEN ASM_REWRITE_TAC[]
    ;
      % Subgoal 2: induction step %
      REWRITE_TAC [NTH_TIME_TRUE]
      THEN REPEAT STRIP_TAC
      THEN RES_TAC
      THEN RES_TAC
      THEN EXISTS_TAC "tp'':num"
      THEN ASM_REWRITE_TAC[]
      THEN IMP_RES_TAC LESS_EQ_STABLE_FALSE_THEN_TRUE
      THEN SUBGOAL_THEN
             "STABLE_FALSE_THEN_TRUE Ee (tp''+1,te')"
             ASSUME_TAC
      THENL [
        % Subgoal 2.1: New subgoal %
        IMP_RES_TAC
          (SPECL ["t':num";"tp'':num";"1"] (SYM_RULE LESS_EQ_MONO_ADD_EQ))
        THEN ASM_CASES_TAC "(tp''+1) <= te'"
        THENL [IMP_RES_TAC SUB_STABLE_FALSE_THEN_TRUE;ALL_TAC]
        THEN IMP_RES_TAC NOT_LESS_EQ_LESS
        THEN IMP_RES_TAC (REWRITE_RULE [ADD1] LT_SUC_IMP_LE)
        THEN IMP_RES_TAC (REWRITE_RULE [ADD1] SUC_LE_IMP_LT)
        THEN FILTER_POP_ASSUM
               (\tm. tm = "TRUE_THEN_STABLE_FALSE Ee(t',tp'')")
               (\thm. STRIP_ASSUME_TAC (REWRITE_RULE[TRUE_THEN_STABLE_FALSE]thm))
        THEN FILTER_POP_ASSUM
               (\tm. tm = "!t. t' < t /\ t <= tp'' ==> ~Ee t")
```

```
            (\thm. STRIP_ASSUME_TAC (SPEC "te':time" thm))
        THEN FILTER_POP_ASSUM
            (\tm. tm = "STABLE_FALSE_THEN_TRUE Ee(t' + 1,te')")
            (\thm. STRIP_ASSUME_TAC(REWRITE_RULE[STABLE_FALSE_THEN_TRUE]thm))
        THEN RES_TAC
      ;
        % Continue %
        RES_TAC
        THEN ASM_CASES_TAC "(tp''+1) <= te'-1"
        THENL [
          IMP_RES_TAC (REWRITE_RULE [ADD1] SUC_LE_IMP_LT)
          THEN IMP_RES_TAC THEN_STABLE_FALSE
          THEN IMP_RES_TAC SUP_INTERVAL_STABLE_FALSE_THEN_TRUE
          THEN ASSUME_TAC
                 (SPECL ["1";"t':time"]
                   (PURE_ONCE_REWRITE_RULE [ADD_SYM] LESS_EQ_ADD))
          THEN IMP_RES_TAC LESS_EQ_TRANS
          THEN IMP_RES_TAC (REWRITE_RULE [PRE_SUB1] LE_PRE_IMP_LT)
          THEN IMP_RES_TAC LESS_LESS_EQ_TRANS
          THEN IMP_RES_TAC LT_IMP_LE
          THEN IMP_RES_TAC SUB_ADD
          THEN FILTER_POP_ASSUM
                 (\tm. tm = "(te' - 1) + 1 = te'")
                 (\thm. RULE_ASSUM_TAC (REWRITE_RULE [thm;LESS_EQ_REFL]))
          THEN RES_TAC
        ;
          IMP_RES_TAC NOT_LESS_EQ_LESS
          THEN IMP_RES_TAC (REWRITE_RULE [ADD1] LT_IMP_SUC_LE)
          THEN ASSUME_TAC
                 (SPECL ["1";"t':time"]
                   (PURE_ONCE_REWRITE_RULE [ADD_SYM] LESS_EQ_ADD))
          THEN IMP_RES_TAC LESS_EQ_TRANS
          THEN IMP_RES_TAC SUB_ADD
          THEN FILTER_POP_ASSUM
                 (\tm. tm = "(te' - 1) + 1 = te'")
                 (\thm. RULE_ASSUM_TAC (REWRITE_RULE [thm;LESS_EQ_REFL]))
          THEN IMP_RES_TAC LESS_EQ_STABLE_FALSE_THEN_TRUE
          THEN IMP_RES_TAC LESS_EQ_TRANS
          THEN IMP_RES_TAC SUB_STABLE_FALSE_THEN_TRUE
        ]
      ]
    ]
   );;

close_theory();;
```

B.3 Multirate Injective Interpreters

```
%-----------------------------------------------------------------------------

   Filename:     in_intrp.ml

   Author:       (c) D.A. Fura 1993

   Date:         30 September 1993

   Test-case theory for injective multirate interpreters.
   Four major theorems:
     (a) H interpreter implements multirate H interpreter.
     (b) Concrete structure implements abstract structure.
     (c) Abstract structure implements multirate GH interpreter.
     (d) Multirate GH interpreter implements GH interpreter.

-----------------------------------------------------------------------------%

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/hol/ml/';
                                  '/home/elvis6/dfura/hol/Library/tools/'
                                 ]);;
```

```
set_flag ('timing',true);;

system 'rm in_intrp.th';;

new_theory 'in_intrp';;

load_library 'reduce';;

loadf 'aux_defs';;
loadf 'abstract';;
loadf 'pt_tacs';;

map load_parent ['ineq';'templogic_def'];;

new_type_abbrev ('time',":num");;
new_type_abbrev ('time'',":num");;

let REP_lemma = new_abstract_representation
   [('Ge', ":(time'->*io)->time'->bool");
    ('Gp', ":(time'->*io)->time'->bool");
    ('He', ":(time'->*io)->time'->bool");
    ('Hp', ":(time'->*io)->time'->bool")
   ];;

make_inst_thms REP_lemma;;

let REP = abstract_type 'in_intrp' 'Ge';;

%-------------------------------------------------------------
   Liveness assumptions.
-------------------------------------------------------------%

let GE_LIVE = new_definition
   ('GE_LIVE',
    "! (rep:^REP) (eg':time'->*io) .
     GE_LIVE rep eg' =
       !t. ?t'. NTH_TIME_TRUE t (Ge rep eg') 0 t'"
   );;

let GP_LIVE = new_definition
   ('GP_LIVE',
    "! (rep:^REP) (eg':time'->*io) (pg':time'->*io) .
     GP_LIVE rep eg' pg' =
       !t t'.
       NTH_TIME_TRUE t (Ge rep eg') 0 t' ==> NTH_TIME_TRUE t (Gp rep pg') 0 t'"
   );;

let HE_LIVE = new_definition
   ('HE_LIVE',
    "! (rep:^REP) (pg':time'->*io) (eh':time'->*io) .
     HE_LIVE rep pg' eh' =
       !t t'.
       NTH_TIME_TRUE t (Gp rep pg') 0 t' ==> NTH_TIME_TRUE t (Gp rep eh') 0 t'"
   );;

let HE_LIVE1 = new_definition
   ('HE_LIVE1',
    "! (rep:^REP) (eh':time'->*io) .
     HE_LIVE1 rep eh' =
       !t t'.
       NTH_TIME_TRUE t (Gp rep eh') 0 t'
       ==> ?u. NTH_TIME_TRUE u (He rep eh') 0 t'"
   );;

let HP_LIVE = new_definition
   ('HP_LIVE',
    "! (rep:^REP) (eh':time'->*io) (ph':time'->*io) .
     HP_LIVE rep eh' ph' =
       !u t'.
       NTH_TIME_TRUE u (He rep eh') 0 t' ==> NTH_TIME_TRUE u (Hp rep ph') 0 t'"
   );;
```

```
%-------------------------------------------------------------
   Concrete interpreters.
-------------------------------------------------------------%

let G_INTRP' = new_definition
   ('G_INTRP'',
    "! (eg':time'->*io) (pg':time'->*io) .
     G_INTRP' eg' pg' = !t'. pg' t' = eg' t'"
   );;

let H_INTRP' = new_definition
   ('H_INTRP'',
    "! (eh':time'->*io) (ph':time'->*io) .
     H_INTRP' eh' ph' = !t'. ph' t' = eh' t'"
   );;

%-------------------------------------------------------------
   Abstraction and abstract interpreters.
-------------------------------------------------------------%

let G_ABS = new_definition
   ('G_ABS',
    "! (rep:^REP) (eg:time->*io) (pg:time->*io) (eg':time->*io) (pg':time->*io) .
     G_ABS rep eg pg eg' pg' =
       !t.
       let t' = @t'. NTH_TIME_TRUE t (Ge rep eg') 0 t' in
       let t'' = @t'. NTH_TIME_TRUE t (Gp rep pg') 0 t' in
       ((eg t = eg' t') /\
        (pg t = pg' t''))"
   );;

let H_ABS = new_definition
   ('H_ABS',
    "! (rep:^REP) (eh:time->*io) (ph:time->*io) (eh':time->*io) (ph':time->*io) .
     H_ABS rep eh ph eh' ph' =
       !u.
       let t' = @t'. NTH_TIME_TRUE u (He rep eh') 0 t' in
       let t'' = @t'. NTH_TIME_TRUE u (Hp rep ph') 0 t' in
       ((eh u = eh' t') /\
        (ph u = ph' t''))"
   );;

let G_INTRP = new_definition
   ('G_INTRP',
    "! (eg:time->*io) (pg:time->*io) .
     G_INTRP eg pg = !t. pg t = eg t"
   );;

let H_INTRP = new_definition
   ('H_INTRP',
    "! (eh:time->*io) (ph:time->*io) .
     H_INTRP eh ph = !u. ph u = eh u"
   );;

%-------------------------------------------------------------
   Interpreter correctness assumptions.
-------------------------------------------------------------%

let G_INTRP_CORR = new_definition
   ('G_INTRP_CORR',
    "! (rep:^REP) .
     G_INTRP_CORR rep =
       !(eg:time->*io) (pg:time->*io) (eg':time'->*io) (pg':time'->*io) .
       G_INTRP' eg' pg' ==>
        G_ABS rep eg pg eg' pg' ==>
         G_INTRP eg pg"
   );;

let H_INTRP_CORR = new_definition
   ('H_INTRP_CORR',
    "! (rep:^REP) .
     H_INTRP_CORR rep =
```

```
       !(eh:time->*io) (ph:time->*io) (eh':time'->*io) (ph':time'->*io) .
       H_INTRP' eh' ph' ==>
        H_ABS rep eh ph eh' ph' ==>
         H_INTRP eh ph"
   );;

%-------------------------------------------------------------
   Multirate interpreter definition and correctness proof.
-------------------------------------------------------------%

let H1_INTRP = new_definition
   ('H1_INTRP',
    "! (rep:^REP) (eh:time->*io) (ph:time->*io) (eh':time'->*io)
       (ph':time'->*io) .
     H1_INTRP rep eh ph eh' ph' =
       !t.
       let t' = @t'. NTH_TIME_TRUE t (Gp rep eh') 0 t' in
       let u = @u. NTH_TIME_TRUE u (Hp rep ph') 0 t' in
       (ph u = eh t)"
   );;

let H1_ABS = new_definition                              /
   ('H1_ABS',
    "! (rep:^REP) (eh:time->*io) (ph:time->*io) (eh':time->*io) (ph':time->*io) .
     H1_ABS rep eh ph eh' ph' =
       !t u.
       let t' = @t'  NTH_TIME_TRUE t (Gp rep eh') 0 t' in
       let t'' = @t'. NTH_TIME_TRUE u (Hp rep ph') 0 t' in
       ((eh t = eh' t') /\
        (ph u = ph' t''))"
   );;

let SELECT_ELIM_TAC tm thm :tactic =
   SUBGOAL_THEN
     tm (\thm. (REWRITE_TAC [thm] THEN RULE_ASSUM_TAC(REWRITE_RULE[thm])))
   THENL [
     SELECT_UNIQUE_TAC
     THEN ASM_REWRITE_TAC[]
     THEN REPEAT STRIP_TAC
     THEN IMP_RES_TAC thm
   ;
     ALL_TAC
   ];;

let H1_INTRP_THM = TAC_PROOF
   (([],
     "!(rep:^REP) (eh:time->*io) (ph:time->*io) (eh':time'->*io)
       (ph':time'->*io) (eg':time'->*io) (pg':time'->*io) .
      H_INTRP eh ph ==>
       GE_LIVE rep eg' ==>
        GP_LIVE rep eg' pg' ==>
         HE_LIVE rep pg' eh' ==>
          HE_LIVE1 rep eh' ==>
           HP_LIVE rep eh' ph' ==>
            H_ABS rep eh ph eh' ph' ==>
             H1_ABS rep eh ph eh' ph' ==>
              H1_INTRP rep eh ph eh' ph'"),
    REWRITE_TAC
      [H_INTRP;H1_INTRP;GE_LIVE;GP_LIVE;HE_LIVE;HE_LIVE1;HP_LIVE;H_ABS;H1_ABS]
    THEN EXPAND_LET_TAC
    THEN REPEAT STRIP_TAC
    THEN FILTER_POP_ASSUM
           (\tm. tm = "!t. ?t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t'")
           (\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN SELECT_ELIM_TAC
           "(@t'. NTH_TIME_TRUE t(Gp rep (eh':time'->*io))0 t') = t'"
           TRUE_EVENT_TIMES_EQUAL
    THEN SELECT_ELIM_TAC
```

```
        "(@u. NTH_TIME_TRUE u(Hp rep (ph':time'->*io))0 t') = u"
        TRUE_EVENT_COUNTS_EQUAL
  THEN RULE_ASSUM_TAC (\thm. SPEC_ALL thm)
  THEN SELECT_ELIM_TAC
        "(@t'. NTH_TIME_TRUE u(He rep (eh':time'->*io))0 t') = t'"
        TRUE_EVENT_TIMES_EQUAL
  THEN SELECT_ELIM_TAC
        "(@t'. NTH_TIME_TRUE t(Gp rep (eh':time'->*io))0 t') = t'"
        TRUE_EVENT_TIMES_EQUAL
  THEN FILTER_POP_ASSUM
        (\tm. tm = "(ph:time->*io) u = (eh:time->*io) u")
        (\thm. REWRITE_TAC[thm])
  THEN ASM_REWRITE_TAC[]
 );;

%-------------------------------------------------------------
   Structure definitions and correctness proof.
-------------------------------------------------------------%

let GH_STRUCT' = new_definition
   ('GH_STRUCT'',
    "! (rep:^REP) (eg':time'->*io) (ph':time'->*io) (eg:time->*io)
       (ph:time->*io)  .
      GH_STRUCT' rep eg' ph' eg ph =
        ?gh' pg eh .
        G_INTRP' eg' gh' /\
        H_INTRP' gh' ph' /\
        G_ABS rep eg pg eg' gh' /\
        H1_ABS rep eh ph gh' ph'"
   );;

let GH_STRUCT = new_definition
   ('GH_STRUCT',
    "! (rep:^REP) (eg:time->*io) (ph:time->*io) (eh':time'->*io)
       (ph':time'->*io) .
      GH_STRUCT rep eg ph eh' ph' =
        ?gh.
        G_INTRP eg gh /\
        H1_INTRP rep gh ph eh' ph'"
   );;

let GH_STRUCT_THM = TAC_PROOF
   (([],
     "!(rep:^REP) (eg:time->*io) (ph:time->*io) (eg':time'->*io) (ph':time'->*io)
       (pg:time->*io) (eh:time->*io) (gh':time'->*io) .
       G_INTRP' eg' gh' /\
       H_INTRP' gh' ph' /\
       G_ABS rep eg pg eg' gh' /\
       H1_ABS rep eh ph gh' ph' /\
       G_INTRP_CORR rep /\
       H_INTRP_CORR rep /\
       GE_LIVE rep eg' /\
       GP_LIVE rep eg' gh' /\
       HE_LIVE rep gh' gh' /\
       HE_LIVE1 rep gh' /\
       HP_LIVE rep gh' ph' /\
       G_ABS rep eg pg eg' gh' /\
       H_ABS rep eh ph gh' ph' /\
       H1_ABS rep eh ph gh' ph' ==>
       GH_STRUCT rep eg ph gh' ph'"),
    REWRITE_TAC [GH_STRUCT;G_INTRP_CORR;H_INTRP_CORR]
    THEN REPEAT STRIP_TAC
    THEN RES_TAC
    THEN IMP_RES_TAC H1_INTRP_THM
    THEN POP_ASSUM_LIST
           (MAP_EVERY
             (\thm. STRIP_ASSUME_TAC
                    (EXPAND_LET_RULE
                     (REWRITE_RULE [G_INTRP';H_INTRP';
                                    G_INTRP_CORR;H_INTRP_CORR;G_ABS;
                                    H_ABS;H1_ABS;GH_STRUCT;HP_LIVE;HE_LIVE;
                                    HE_LIVE1;GP_LIVE;GE_LIVE] thm))))
```

```
    THEN FILTER_POP_ASSUM
           (\tm. tm ="!t  ?t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t'")
           (\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN SUBGOAL_THEN
           "(pg:time->*io) = eh"
           ASSUME_TAC
    THENL [
      CONV_TAC (ONCE_DEPTH_CONV FUN_EQ_CONV)
      THEN GEN_TAC
      THEN ASM_REWRITE_TAC[]
    ;
      EXISTS_TAC "(pg:time->*io)"
      THEN ASM_REWRITE_TAC[]
    ]
   );;

%-------------------------------------------------------------
   Multirate system interpreter definition and proof.
-------------------------------------------------------------%

let GH1_INTRP = new_definition
   ('GH1_INTRP',
    "! (rep:^REP) (eg:time->*io) (ph:time->*io) (eg':time'->*io)
       (ph':time'->*io) .
     GH1_INTRP rep eg ph eg' ph' =
       !t.
       let t' = @t'. NTH_TIME_TRUE t (Ge rep eg') 0 t' in
       let u = @u. NTH_TIME_TRUE u (Hp rep ph') 0 t' in
       (ph u = eg t)"
   );;

let GH1_INTRP_THM = TAC_PROOF
   (([],
     "! (rep:^REP) (eg:time->*io) (ph:time->*io) (eg':time'->*io)
        (pg':time'->*io) (eh':time'->*io) (ph':time'->*io)
      GH_STRUCT rep eg ph eh' ph' ==>
      GE_LIVE rep eg' ==>
      GP_LIVE rep eg' pg' ==>
      HE_LIVE rep pg' eh' ==>
      GH1_INTRP rep eg ph eg' ph'"),
    REWRITE_TAC [GH_STRUCT;GH1_INTRP;G_INTRP;H1_INTRP;GE_LIVE;GP_LIVE;HE_LIVE]
    THEN EXPAND_LET_TAC
    THEN REPEAT STRIP_TAC
    THEN FILTER_POP_ASSUM
           (\tm. tm = "!t. (gh:time->*io) t = (eg:time->*io) t")
           (\thm. REWRITE_TAC [SYM_RULE thm])
    THEN FILTER_POP_ASSUM
           (\tm. tm = "!t. ?t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t'")
           (\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
    THEN RES_TAC
    THEN RES_TAC
    THEN SELECT_ELIM_TAC
           "(@t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t') = t'"
           TRUE_EVENT_TIMES_EQUAL
    THEN RULE_ASSUM_TAC (\thm. SPEC_ALL thm)
    THEN SELECT_ELIM_TAC
           "(@t'. NTH_TIME_TRUE t(Gp rep (eh':time'->*io))0 t') = t'"
           TRUE_EVENT_TIMES_EQUAL
    THEN ASM_REWRITE_TAC[]
   );;

%-------------------------------------------------------------
   Final system interpreter definition and proof.
-------------------------------------------------------------%

let GH_INTRP = new_definition
   ('GH_INTRP',
    "! (eg:time->*io) (ph:time->*io) .
```

```
     GH_INTRP eg ph =
       !t. ph t = eg t"
   );;

let GH_ABS = new_definition
   ('GH_ABS',
    "! (rep:^REP) (eg:time->*io) (ph:time->*io) (eg':time->*io) (ph':time->*io) .
     GH_ABS rep eg ph eg' ph' =
       !t.
       let t' = @t'. NTH_TIME_TRUE t (Ge rep eg') 0 t' in
       ((eg t = eg' t') /\
        (ph t = ph' t'))"
   );;

let GH_INTRP_THM = TAC_PROOF
   (([],
     "! (rep:^REP) (eg:time->*io) (ph:time->*io) (eg':time'->*io)
        (pg':time'->*io) (eh':time'->*io) (ph':time'->*io) .
      GH1_INTRP rep eg ph eg' ph' ==>
      GE_LIVE rep eg' ==>
      GP_LIVE rep eg' pg' ==>
      HE_LIVE rep pg' eh' ==>
      HE_LIVE1 rep eh' ==>
      HP_LIVE rep eh' ph' ==>
      H1_ABS rep eh ph eh' ph' ==>
      GH_ABS rep eg ph eg' ph' ==>
      GH_INTRP eg ph"),
    REWRITE_TAC [GH1_INTRP;GH_INTRP;GE_LIVE;GP_LIVE;HE_LIVE;HE_LIVE1;
                 HP_LIVE;GH_ABS;H1_ABS]
    THEN EXPAND_LET_TAC
    THEN REPEAT STRIP_TAC
    THEN FILTER_POP_ASSUM
           (\tm. tm = "!t. ?t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t'")
           (\thm. STRIP_ASSUME_TAC (SPEC_ALL thm))
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN RES_TAC
    THEN RULE_ASSUM_TAC SPEC_ALL
    THEN SELECT_ELIM_TAC
           "(@t'. NTH_TIME_TRUE t(Ge rep (eg':time'->*io))0 t') = t'"
           TRUE_EVENT_TIMES_EQUAL
    THEN SELECT_ELIM_TAC
           "(@u. NTH_TIME_TRUE u(Hp rep (ph':time'->*io))0 t') = u"
           TRUE_EVENT_COUNTS_EQUAL
    THEN SELECT_ELIM_TAC
           "(@t''. NTH_TIME_TRUE u(Hp rep (ph':time'->*io))0 t'') = t'"
           TRUE_EVENT_TIMES_EQUAL
    THEN FILTER_POP_ASSUM
           (\tm. tm = "(ph:time->*io) u = (eg:time->*io) t")
           (\thm. REWRITE_TAC[SYM_RULE thm])
    THEN ASM_REWRITE_TAC[]
   );;

close_theory();;
```

Appendix C: Processor-Memory Module Specification Examples

This section contains HOL theories for parts of the PMM transaction-level specification. Section C.1 contains the signal data structure definitions for the transaction level. Sections C.2, C.3, and C.4 contain the specifications for the local processor, local memory, and M-Port, respectively.

C.1 Transaction Signal Data Structure Definitions

```
%-----------------------------------------------------------------------------

   File:     pmmtauxp_def.ml

   Author:   (c) D.A. Fura 1993

   Date:     14 September 1993

   This file contains types and definitions for the transaction-level
   specification of the P Process of the FTEP PMM.

-----------------------------------------------------------------------------%

set_flag ('timing',true);;

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/ftep/piu/hol/pport/';
                                  '/home/elvis6/dfura/hol/Library/tools/'
                                 ]);;

system 'rm pmmtauxp_def.th';;

new_theory 'pmmtauxp_def';;

new_type_abbrev ('time', ":num");;
new_type_abbrev ('timeT', ":num");;
new_type_abbrev ('wordn', ":num->bool");;
new_type_abbrev ('wordnn', ":num->wordn");;

%-----------------------------------------------------------------------------
   Abstract data type for the memory access target.
-----------------------------------------------------------------------------%

let targ_Axiom =
   define_type 'targ_Axiom'
               'targ = LM | PIU | CB';;

%-----------------------------------------------------------------------------
   Abstract data type for the L-Bus Master Packet.
-----------------------------------------------------------------------------%

let pbmop =
   define_type 'pbmop'
               'pbmop = PBM_WrLM | PBM_WrPIU | PBM_WrCB | PBM_RdLM |
                        PBM_RdPIU | PBM_RdCB | PBM_Illegal';;

let pbm_pkt =
   define_type 'pbm_pkt' 'pbm_pkt = PBM pbmop wordn wordnn wordn wordn bool';;

let PBM_Opcode = new_recursive_definition
   false pbm_pkt 'PBM_Opcode'
   "PBM_Opcode (PBM Opcode Addr Data BS BE Lock) = Opcode";;

let PBM_Addr = new_recursive_definition
   false pbm_pkt 'PBM_Addr'
   "PBM_Addr (PBM Opcode Addr Data BS BE Lock) = Addr";;

let PBM_Data = new_recursive_definition
   false pbm_pkt 'PBM_Data'
   "PBM_Data (PBM Opcode Addr Data BS BE Lock) = Data";;
```

```
let PBM_BS = new_recursive_definition
   false pbm_pkt 'PBM_BS'
   "PBM_BS (PBM Opcode Addr Data BS BE Lock) = BS";;

let PBM_BE = new_recursive_definition
   false pbm_pkt 'PBM_BE'
   "PBM_BE (PBM Opcode Addr Data BS BE Lock) = BE";;

let PBM_Lock = new_recursive_definition
   false pbm_pkt 'PBM_Lock'
   "PBM_Lock (PBM Opcode Addr Data BS BE Lock) = Lock";;

let PBM_CASES = prove_cases_thm (prove_induction_thm pbm_pkt);;

let PBM_Selectors_Work = prove_thm
   ('PBM_Selectors_Work',
    "!k:pbm_pkt.
     k = (PBM (PBM_Opcode k) (PBM_Addr k) (PBM_Data k) (PBM_BS k) (PBM_BE k)
              (PBM_Lock k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:pbm_pkt" PBM_CASES)
    THEN REWRITE_TAC [PBM_Opcode; PBM_Addr; PBM_Data; PBM_BS; PBM_BE; PBM_Lock]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the L-Bus Slave Packet.
-----------------------------------------------------------------------------%

let pbsop = define_type 'pbsop' 'pbsop = PBS_Ready | PBS_Illegal';;

let pbs_pkt = define_type 'pbs_pkt' 'pbs_pkt = PBS pbsop wordnn';;

let PBS_Opcode = new_recursive_definition
   false pbs_pkt 'PBS_Opcode' "PBS_Opcode (PBS Opcode Data) = Opcode";;

let PBS_Data = new_recursive_definition
   false pbs_pkt 'PBS_Data' "PBS_Data (PBS Opcode Data) = Data";;

let PBS_CASES = prove_cases_thm (prove_induction_thm pbs_pkt);;

let PBS_Selectors_Work = prove_thm
   ('PBS_Selectors_Work',
    "!k:pbs_pkt.  k = (PBS (PBS_Opcode k) (PBS_Data k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:pbs_pkt" PBS_CASES)
    THEN REWRITE_TAC [PBS_Opcode;PBS_Data]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the I-Bus Slave Packet.
-----------------------------------------------------------------------------%

let ibsop = define_type 'ibsop' 'ibsop = IBS_Ready | IBS_Idle | IBS_Illegal';;

let ibs_pkt = define_type 'ibs_pkt' 'ibs_pkt = IBS ibsop wordnn';;

let IBS_Opcode = new_recursive_definition
   false ibs_pkt 'IBS_Opcode' "IBS_Opcode (IBS Opcode Data) = Opcode";;

let IBS_Data = new_recursive_definition
   false ibs_pkt 'IBS_Data' "IBS_Data (IBS Opcode Data) = Data";;

let IBS_CASES = prove_cases_thm (prove_induction_thm ibs_pkt);;

let IBS_Selectors_Work = prove_thm
   ('IBS_Selectors_Work',
    "!k:ibs_pkt.  k = (IBS (IBS_Opcode k) (IBS_Data k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:ibs_pkt" IBS_CASES)
    THEN REWRITE_TAC [IBS_Opcode;IBS_Data]
   );;
```

```
%-----------------------------------------------------------------------------
   Abstract data type for the I-Bus Arbitration Master Packet.
-----------------------------------------------------------------------------%

let ibamop = define_type 'ibamop' 'ibamop = IBAM_Ready | IBAM_Illegal';;

let ibam_pkt = define_type 'ibam_pkt' 'ibam_pkt = IBAM ibamop';;

let IBAM_Opcode = new_recursive_definition
   false ibam_pkt 'IBAM_Opcode' "IBAM_Opcode (IBAM Opcode) = Opcode";;

let IBAM_CASES = prove_cases_thm (prove_induction_thm ibam_pkt);;

let IBAM_Selectors_Work = prove_thm
   ('IBAM_Selectors_Work',
    "!k:ibam_pkt.  k = (IBAM (IBAM_Opcode k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:ibam_pkt" IBAM_CASES)
    THEN REWRITE_TAC [IBAM_Opcode]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the I-Bus Arbitration Slave Packet.
-----------------------------------------------------------------------------%

let ibasop = define_type 'ibasop' 'ibasop = IBAS_Ready | IBAS_Illegal';;

let ibas_pkt = define_type 'ibas_pkt' 'ibas_pkt = IBAS ibasop';;

let IBAS_Opcode = new_recursive_definition
   false ibas_pkt 'IBAS_Opcode' "IBAS_Opcode (IBAS Opcode) = Opcode";;

let IBAS_CASES = prove_cases_thm (prove_induction_thm ibas_pkt);;

let IBAS_Selectors_Work = prove_thm
   ('IBAS_Selectors_Work',
    "!k:ibas_pkt.  k = (IBAS (IBAS_Opcode k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:ibas_pkt" IBAS_CASES)
    THEN REWRITE_TAC [IBAS_Opcode]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the M-Bus Master Packet.
-----------------------------------------------------------------------------%

let mbmop =
   define_type 'mbmop' 'mbmop = MBM_Wr | MBM_Rd | MBM_RdWr | MBM_Illegal';;

let mbm_pkt =
   define_type 'mbm_pkt' 'mbm_pkt = MBM mbmop wordn wordnn wordn';;

let MBM_Opcode = new_recursive_definition
   false mbm_pkt 'MBM_Opcode'
   "MBM_Opcode (MBM Opcode Addr Data BS) = Opcode";;

let MBM_Addr = new_recursive_definition
   false mbm_pkt 'MBM_Addr'
   "MBM_Addr (MBM Opcode Addr Data BS) = Addr";;

let MBM_Data = new_recursive_definition
   false mbm_pkt 'MBM_Data'
   "MBM_Data (MBM Opcode Addr Data BS) = Data";;

let MBM_BS = new_recursive_definition
   false mbm_pkt 'MBM_BS'
   "MBM_BS (MBM Opcode Addr Data BS) = BS";;

let MBM_CASES = prove_cases_thm (prove_induction_thm mbm_pkt);;

let MBM_Selectors_Work = prove_thm
   ('MBM_Selectors_Work',
```

```
    "!k:mbm_pkt.
     k = (MBM (MBM_Opcode k) (MBM_Addr k) (MBM_Data k) (MBM_BS k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:mbm_pkt" MBM_CASES)
    THEN REWRITE_TAC [MBM_Opcode;MBM_Addr;MBM_Data;MBM_BS]
   );;

%----------------------------------------------------------------------------
   Abstract data type for the M-Bus Slave Packet.
----------------------------------------------------------------------------%

let mbsop = define_type 'mbsop' 'mbsop = MBS_Ready | MBS_Illegal';;

let mbs_pkt = define_type 'mbs_pkt' 'mbs_pkt = MBS mbsop wordnn';;

let MBS_Opcode = new_recursive_definition
   false mbs_pkt 'MBS_Opcode' "MBS_Opcode (MBS Opcode Data) = Opcode";;

let MBS_Data = new_recursive_definition
   false mbs_pkt 'MBS_Data' "MBS_Data (MBS Opcode Data) = Data";;

let MBS_CASES = prove_cases_thm (prove_induction_thm mbs_pkt);;

let MBS_Selectors_Work = prove_thm
   ('MBS_Selectors_Work',
    "!k:mbs_pkt.  k = (MBS (MBS_Opcode k) (MBS_Data k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:mbs_pkt" MBS_CASES)
    THEN REWRITE_TAC [MBS_Opcode;MBS_Data]
   );;


%----------------------------------------------------------------------------
   Abstract data type for the C-Bus Master Packet.
----------------------------------------------------------------------------%

let cbmop =
   define_type 'cbmop' 'cbmop = CBM_Wr | CBM_Rd | CBM_Illegal';;

let cbm_pkt =
   define_type 'cbm_pkt' 'cbm_pkt = CBM cbmop wordn wordnn wordn wordn';;

let CBM_Opcode = new_recursive_definition
   false cbm_pkt 'CBM_Opcode'
   "CBM_Opcode (CBM Opcode Addr Data BS BE) = Opcode";;

let CBM_Addr = new_recursive_definition
   false cbm_pkt 'CBM_Addr'
   "CBM_Addr (CBM Opcode Addr Data BS BE) = Addr";;

let CBM_Data = new_recursive_definition
   false cbm_pkt 'CBM_Data'
   "CBM_Data (CBM Opcode Addr Data BS BE) = Data";;

let CBM_BS = new_recursive_definition
   false cbm_pkt 'CBM_BS'
   "CBM_BS (CBM Opcode Addr Data BS BE) = BS";;

let CBM_BE = new_recursive_definition
   false cbm_pkt 'CBM_BE'
   "CBM_BE (CBM Opcode Addr Data BS BE) = BE";;

let CBM_CASES = prove_cases_thm (prove_induction_thm cbm_pkt);;

let CBM_Selectors_Work = prove_thm
   ('CBM_Selectors_Work',
    "!k:cbm_pkt.
     k = (CBM (CBM_Opcode k) (CBM_Addr k) (CBM_Data k) (CBM_BS k) (CBM_BE k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:cbm_pkt" CBM_CASES)
    THEN REWRITE_TAC [CBM_Opcode;CBM_Addr;CBM_Data;CBM_BS;CBM_BE]
   );;
```

```
%-----------------------------------------------------------------------------
   Abstract data type for the C-Bus Slave Packet.
-----------------------------------------------------------------------------%

let cbsop = define_type 'cbsop' 'cbsop = CBS_Ready | CBS_Illegal';;

let cbs_pkt = define_type 'cbs_pkt' 'cbs_pkt = CBS cbsop wordnn';;

let CBS_Opcode = new_recursive_definition
   false cbs_pkt 'CBS_Opcode' "CBS_Opcode (CBS Opcode Data) = Opcode";;

let CBS_Data = new_recursive_definition
   false cbs_pkt 'CBS_Data' "CBS_Data (CBS Opcode Data) = Data";;

let CBS_CASES = prove_cases_thm (prove_induction_thm cbs_pkt);;

let CBS_Selectors_Work = prove_thm
   ('CBS_Selectors_Work',
    "!k:cbs_pkt.  k = (CBS (CBS_Opcode k) (CBS_Data k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:cbs_pkt" CBS_CASES)
    THEN REWRITE_TAC [CBS_Opcode;CBS_Data]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the P-Port, M-Port, and R-Port Reset Master Packet.
-----------------------------------------------------------------------------%

let rmop = define_type 'rmop' 'rmop = RM_NoReset | RM_Illegal';;

let rm_pkt = define_type 'rm_pkt' 'rm_pkt = RM rmop';;

let RM_Opcode = new_recursive_definition
   false rm_pkt 'RM_Opcode' "RM_Opcode (RM Opcode) = Opcode";;

let RM_CASES = prove_cases_thm (prove_induction_thm rm_pkt);;

let RM_Selectors_Work = prove_thm
   ('RM_Selectors_Work',
    "!k:rm_pkt.  k = (RM (RM_Opcode k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:rm_pkt" RM_CASES)
    THEN REWRITE_TAC [RM_Opcode]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the C-Port Reset Master Packet.
-----------------------------------------------------------------------------%

let crmop = define_type 'crmop' 'crmop = CRM_NoReset | CRM_Illegal';;

let crm_pkt = define_type 'crm_pkt' 'crm_pkt = CRM crmop';;

let CRM_Opcode = new_recursive_definition
   false crm_pkt 'CRM_Opcode' "CRM_Opcode (CRM Opcode) = Opcode";;

let CRM_CASES = prove_cases_thm (prove_induction_thm crm_pkt);;

let CRM_Selectors_Work = prove_thm
   ('CRM_Selectors_Work',
    "!k:crm_pkt.  k = (CRM (CRM_Opcode k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:crm_pkt" CRM_CASES)
    THEN REWRITE_TAC [CRM_Opcode]
   );;

%-----------------------------------------------------------------------------
   Abstract data type for the External Reset Master Packet.
-----------------------------------------------------------------------------%

let ermop = define_type 'ermop' 'ermop = ERM_NoReset | ERM_Illegal';;
```

```
let erm_pkt = define_type 'erm_pkt' 'erm_pkt = ERM ermop';;

let ERM_Opcode = new_recursive_definition
   false erm_pkt 'ERM_Opcode' "ERM_Opcode (ERM Opcode) = Opcode";;

let ERM_CASES = prove_cases_thm (prove_induction_thm erm_pkt);;

let ERM_Selectors_Work = prove_thm
   ('ERM_Selectors_Work',
    "!k:erm_pkt.  k = (ERM (ERM_Opcode k))",
    GEN_TAC
    THEN STRUCT_CASES_TAC (SPEC "k:erm_pkt" ERM_CASES)
    THEN REWRITE_TAC [ERM_Opcode]
   );;

close_theory();;
```

C.2 Local Processor Transaction-Level Model

```
%-----------------------------------------------------------------------------

   File:      cputransp_def.ml

   Author:    (c) D.A. Fura 1993

   Date:      21 April 1993

   This file contains the ml source for the trans-level specification of the
   CPU of the FTEP PMM, a processing module developed by the Embedded
   Processing Laboratory, Boeing High Technology Center.

-----------------------------------------------------------------------------%

set_flag ('timing',true);;

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/pport/';
                                  '/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/ftep/piu/hol/pmm/';
                                  '/home/elvis6/dfura/hol/Library/tools/'
                                 ]);;

system 'rm cputransp_def.th';;

new_theory 'cputransp_def';;

map new_parent ['pmmtauxp_def';'cputauxp_def';'array_def';'wordn_def'];;

new_type_abbrev ('wordn', ":num->bool");;
new_type_abbrev ('wordnn', ":num->wordn");;

%-----------------------------------------------------------------------------
   CPU interpreter definition.
-----------------------------------------------------------------------------%

let CPUT_EXEC = new_definition
   ('CPUT_EXEC',
    "! (cputi :CPUTI) (s :timeT->cput_state) (er :timeT->rm_pkt)
       (eq :timeT->cpurm_pkt)(el :timeT->pbs_pkt)(p :timeT->pbm_pkt) (t :timeT) .
     CPUT_EXEC cputi s (er, eq, el) p t =
        (RM_Opcode (er t) = RM_NoReset) /\
        (CPURM_Opcode (eq t) = CPURM_Rqt) /\
        (PBS_OpRdy (el t) = PBS_Rdy)"
   );;

let CPUT_PRE = new_definition
   ('CPUT_PRE',
    "! (cputi :CPUTI) (s :timeT->cput_state) (er :timeT->rm_pkt)
       (eq :timeT->cpurm_pkt)(el :timeT->pbs_pkt)(p :timeT->pbm_pkt) (t :timeT) .
     CPUT_PRE cputi s (er, eq, el) p t = T"
```

```
   );;

let CPUT_POST = new_definition
   ('CPUT_POST',
    "! (cputi :CPUTI) (s :timeT->cput_state) (er :timeT->rm_pkt)
       (eq :timeT->cpurm_pkt)(el :timeT->pbs_pkt)(p :timeT->pbm_pkt) (t :timeT)
     CPUT_POST cputi s (er, eq, el) p t =
       let lmem = (~ELEMENT (CPUT_addrS (s t)) 31) /\
                  (~(SUBARRAY (CPUT_addrS (s t)) (25,24) = WORDN 1 3)) in
       let piu = (~ELEMENT (CPUT_addrS (s t)) 31) /\
                 (SUBARRAY (CPUT_addrS (s t)) (25,24) = WORDN 1 3) in
       let cbus = ELEMENT (CPUT_addrS (s t)) 31 in
       let write = CPUT_wrS (s t) in
       ((PBM_Opcode (p t) = lmem =>  (write => PBM_WrLM | PBM_RdLM) |
                            piu =>   (write => PBM_WrPIU | PBM_RdPIU)
                          % cbus % | (write => PBM_WrCB | PBM_RdCB)) /\
        (PBM_Addr (p t) = CPUT_addrS (s t)) /\
        (write ==> (PBM_Data (p t) = CPUT_dataS (s t))) /\
        (PBM_BS (p t) = CPUT_bsS (s t)) /\
        (PBM_BE (p t) = CPUT_be_S (s t)) /\
        (PBM_Lock (p t) = CPUT_lock_S (s t)) /\
        (~write ==> (CPUT_dataS (s (t+1)) = PBS_Data (el t))))"
   );;

let CPUT_INSTR = new_definition
   ('CPUT_INSTR',
    "! (cputi :CPUTI) (s :timeT->cput_state) (er :timeT->rm_pkt)
       (eq :timeT->cpurm_pkt) (el :timeT->pbs_pkt)(p :timeT->pbm_pkt) .
     CPUT_INSTR cputi s (er, eq, el) p =
        ! t :timeT .
        CPUT_EXEC cputi s (er, eq, el) p t /\
        CPUT_PRE cputi s (er, eq, el) p t
        ==>
        CPUT_POST cputi s (er, eq, el) p t"
   );;

let CPUT_INTRP = new_definition
   ('CPUT_INTRP',
    "! (s :timeT->cput_state) (er :timeT->rm_pkt) (eq :timeT->cpurm_pkt)
       (el :timeT->pbs_pkt)(p :timeT->pbm_pkt) .
     CPUT_INTRP s (er, eq, el) p =
        ! cputi:CPUTI. CPUT_INSTR cputi s (er, eq, el) p"
   );;

close_theory();;
```

C.3 Local Memory Transaction-Level Model

```
%-----------------------------------------------------------------------------

   File:     memtransp_def.ml

   Author:   (c) D.A. Fura 1993

   Date:     8 September 1993

   This file contains the ml source for the trans-level specification of the
   Local Memory of the FTEP PMM, a processing module developed by the Embedded
   Processing Laboratory, Boeing High Technology Center.

-----------------------------------------------------------------------------%

set_flag ('timing',true);;

set_search_path (search_path() @ ['/home/elvis6/dfura/ftep/piu/hol/lib/';
                                  '/home/elvis6/dfura/ftep/piu/hol/pmm/';
                                  '/home/elvis6/dfura/ftep/piu/hol/mbus/';
```

```
                                                    '/home/elvis6/dfura/ftep/piu/hol/pport/pproc/
';                                    '/home/elvis6/dfura/hol/Library/tools/'
                                   ]);;

system 'rm memtransp_def.th';;

new_theory 'memtransp_def';;

map new_parent ['memtauxp_def';'memaux_def';'pmmtauxp_def';'array_def';
                'wordn_def';'mbtabs_def';'memtabs_def'];;

new_type_abbrev ('wordn', ":num->bool");;
new_type_abbrev ('wordnn', ":num->wordn");;

%----------------------------------------------------------------------------
   M-Bus interpreter definition.
----------------------------------------------------------------------------%

let MEMT_EXEC = new_definition
   ('MEMT_EXEC',
    "! (memti :MEMTI) (s :timeT->memt_state) (e :timeT->mbm_pkt)
       (p :timeT->mbs_pkt) (tm :timeT) .
     MEMT_EXEC memti s e p tm = T"
   );;

let MEMT_PREC = new_definition
   ('MEMT_PREC',
    "! (memti :MEMTI) (s :timeT->memt_state) (e :timeT->mbm_pkt)
       (p :timeT->mbs_pkt) (tm :timeT) .
     MEMT_PREC memti s e p tm = T"
   );;

let MEMT_POSTC = new_definition
   ('MEMT_POSTC',
    "! (memti :MEMTI) (s :timeT->memt_state) (e :timeT->mbm_pkt)
       (p :timeT->mbs_pkt) (tm :timeT) .
     MEMT_POSTC memti s e p tm =
       let adr_min = VAL 23 (MBM_Addr (e tm)) in
       let bs = VAL 1 (MBM_BS (e tm)) in
       let adr_max = adr_min + bs in
       ((MemtS (s (tm+1)) =
           ((MBM_Opcode (e tm) = MBM_Wr) \/ (MBM_Opcode (e tm) = MBM_RdWr))
             => MALTER (MemtS (s tm))
                       (adr_max,adr_min)
                       (MBM_Data (e tm))
              | MemtS (s tm)) /\
        (MBS_Opcode (p tm) = MBS_Ready) /\
        (((MBM_Opcode (e tm) = MBM_Rd) \/ (MBM_Opcode (e tm) = MBM_RdWr))
          ==> (MBS_Data (p tm) = SUBARRAY (MemtS (s tm)) (adr_max,adr_min))))"
   );;

let MEMT_INTRP = new_definition
   ('MEMT_INTRP',
    "! (s :timeT->memt_state) (e :timeT->mbm_pkt) (p :timeT->mbs_pkt) .
     MEMT_INTRP s e p =
        !tm memti.
        MEMT_EXEC memti s e p tm /\
        MEMT_PREC memti s e p tm
        ==>
        MEMT_POSTC memti s e p tm"
   );;

let MEMT_INTRP_ABS = new_definition
   ('MEMT_INTRP_ABS',
    "! (s :timeT->memt_state) (e :timeT->mbm_pkt) (p :timeT->mbs_pkt)
       (s' :timeC->mem_state) (e' :timeC->mb_sin) (p' :timeC->mb_sout) .
     MEMT_INTRP_ABS s e p s' e' p' =
        !tm.
        let tm' = @t'. NTH_TIME_CHANGES_TRUE tm (cs_sig_mbs e') 0 t' in
        let tm'suc = @t'. NTH_TIME_CHANGES_TRUE (SUC tm) (cs_sig_mbs e') 0 t' in
        let tm'done = @t'. STABLE_TRUE_THEN_FALSE (cs_sig_mbs e') (tm',t') in
        (((e tm, p tm) = MB_SLAVE_ABS e' p' (tm', tm'suc)) /\
```

```
        ((s tm) = MEM_ABS s' (tm', tm'done)))"
   );;

let MEMT_INTRP_LIVE = new_definition
   ('MEMT_INTRP_LIVE',
    "! (s :timeT->memt_state) (e :timeT->mbm_pkt) (p :timeT->mbs_pkt)
       (e' :timeC->mb_sin) (p' :timeC->mb_sout) .
     MEMT_INTRP_LIVE s e p e' p' =
        ! (tm:timeT) (memti:MEMTI).
        MEMT_EXEC memti s e p tm
        ==> ?tm'. NTH_TIME_CHANGES_TRUE tm (cs_sig_mbs e') 0 tm' /\ (tm' > 0)"
   );;

close_theory();;
```

C.4 M-Port Transaction-Level Model

```
%-----------------------------------------------------------------------------

   File:      mtransp_def.ml

   Author:    (c) D.A. Fura 1993

   Date:      13 September 1993

   This file contains the ml source for the trans-level specification of the
   M-Port of the FTEP PIU, an ASIC developed by the Embedded Processing
   Laboratory, Boeing High Technology Center.

-----------------------------------------------------------------------------%

set_search_path (search_path() @['/home/elvis6/dfura/ftep/piu/hol/mport/';
                                 '/home/elvis6/dfura/ftep/piu/hol/lib/';
                                 '/home/elvis6/dfura/ftep/piu/hol/mbus/';
                                 '/home/elvis6/dfura/ftep/piu/hol/ibus/';
                                 '/home/elvis6/dfura/ftep/piu/hol/pmm/';
                                 '/home/elvis6/dfura/ftep/piu/hol/pport/pproc/';
                                 '/home/elvis6/dfura/ftep/piu/hol/sucont/pproc/';
                                 '/home/elvis6/dfura/hol/Library/abs_theory/';
                                 '/home/elvis6/dfura/hol/Library/tools/'
                                ]);;

set_flag ('timing',true);;

system 'rm mtransp_def.th';;

new_theory 'mtransp_def';;

loadf 'abstract';;

map new_parent ['pmmtauxp_def';'pmmaux_def';'mtauxp_def';'array_def';'wordn_def';
                'mbtabs_def';'ibtabs_def';'stabs_def';'piuaux_def'];;

let REP = abstract_type 'piuaux_def' 'Andn';;

%-----------------------------------------------------------------------------
   M-Port Interpreter definition
-----------------------------------------------------------------------------%

let MT_EXEC = new_definition
   ('MT_EXEC',
    "! (mti :MTI) (ei :timeT->pbm_pkt) (pm :timeT->mbm_pkt) (em :timeT->mbs_pkt)
       (pi :timeT->ibs_pkt) (er :timeT->rm_pkt) (ti :timeT) .
     MT_EXEC mti (ei,em,er) (pm, pi) ti =
       (RM_Opcode (er ti) = RM_NoReset) /\
       ((mti = MT_Rd) =>  (PBM_Opcode (ei ti) = PBM_RdLM) |
        (mti = MT_Wr) =>  (PBM_Opcode (ei ti) = PBM_WrLM) |
      % (mti = MT_Idle) % ((PBM_Opcode (ei ti) = PBM_RdPIU) \/
                           (PBM_Opcode (ei ti) = PBM_WrPIU) \/
```

```
                        (PBM_Opcode (ei ti) = PBM_RdCB) \/
                        (PBM_Opcode (ei ti) = PBM_WrCB)))"
   );;

let MT_PREC = new_definition
    ('MT_PREC',
     "! (mti :MTI) (ei :timeT->pbm_pkt) (pm :timeT->mbm_pkt) (em :timeT->mbs_pkt)
        (pi :timeT->ibs_pkt) (er :timeT->rm_pkt) (ti :timeT) .
      MT_PREC mti (ei,em,er) (pm,pi) ti = T"
   );;

let MT_POSTC = new_definition
    ('MT_POSTC',
     "! (rep :^REP) (mti :MTI) (ei :timeT->pbm_pkt) (pm :timeT->mbm_pkt)
        (em :timeT->mbs_pkt) (pi :timeT->ibs_pkt) (er :timeT->rm_pkt)
        (ti tm :timeT) .
      MT_POSTC rep mti (ei,em,er) (pm,pi) (ti,tm) =
        let bs = VAL 1 (PBM_BS (ei ti)) in
        ((IBS_Opcode (pi ti) =
           (mti = MT_Rd) => IBS_Ready |
           (mti = MT_Wr) => IBS_Ready |
         %(mti = MT_Idle)% IBS_Idle) /\
         ((mti = MT_Rd)
           ==> (MBS_Opcode (em tm) = MBS_Ready)
                ==> (IBS_Data (pi ti) =
                       MAPN bs (Ham_Dec rep) (MBS_Data (em tm)))) /\
         (~(mti = MT_Idle)
            ==> ((MBM_Opcode (pm tm) =
                    (mti = MT_Rd) => MBM_Rd |
                    ((mti = MT_Wr) /\ (PBM_BE (ei ti) = WORDN 3 15)) => MBM_Wr |
                    MBM_RdWr) /\
                 (MBM_Addr (pm tm) = PBM_Addr (ei ti)) /\
                 ((mti = MT_Wr)
                   ==> (MBM_Data (pm tm) =
                          let be = PBM_BE (ei tm) in
                          MAP_LISTN bs
                            [(Ham_Enc rep) o
                             (\f. BYTE_MUXN 31 be (ELEMENT(PBM_Data(ei ti)) 0) f);
                             (Ham_Enc rep) o
                             (\f. BYTE_MUXN 31 be (ELEMENT(PBM_Data(ei ti)) 1) f);
                             (Ham_Enc rep) o
                             (\f. BYTE_MUXN 31 be (ELEMENT(PBM_Data(ei ti)) 2) f);
                             (Ham_Enc rep) o
                             (\f. BYTE_MUXN 31 be (ELEMENT(PBM_Data(ei ti)) 3) f)]
                               (MAPN bs (Ham_Dec rep) (MBS_Data (em tm))))) /\
                 (MBM_BS (pm tm) = PBM_BS (ei ti)))))"
   );;

let MT_INTRP = new_definition
    ('MT_INTRP',
     "! (rep :^REP) (ei :timeT->pbm_pkt) (pm :timeT->mbm_pkt)
        (em :timeT->mbs_pkt) (pi :timeT->ibs_pkt) (er :timeT->rm_pkt) .
      MT_INTRP rep (ei,em,er) (pm,pi) =
          !ti tm mti.
          MT_EXEC mti (ei,em,er) (pm,pi) ti /\
          MT_PREC mti (ei,em,er) (pm,pi) ti
          ==>
          MT_POSTC rep mti (ei,em,er) (pm,pi) (ti,tm)"
   );;

let MT_INTRP_ABS = new_definition
    ('MT_INTRP_ABS',
     "! (ei :timeT->pbm_pkt) (em :timeT->mbs_pkt) (pm :timeT->mbm_pkt)
        (pi :timeT->ibs_pkt) (er :timeT->rm_pkt) (ei' :timeC->ib_sin)
        (em' :timeC->mb_min) (er' :timeC->su_mout) (pm' :timeC->mb_mout)
        (pi' :timeC->ib_sout) .
      MT_INTRP_ABS (ei,em,er) (pm,pi) (ei',em',er') (pm',pi') =
          !u.
          let ti' = @t'. NTH_TIME_TRUE u (ale_sig_ibs ei') 0 t' in
          let ti'suc = @t'. NTH_TIME_TRUE (SUC u) (ale_sig_ibs ei') 0 t' in
          let tm' = @t'. NTH_TIME_CHANGES_TRUE u (cs_sig_mbm pm') 0 t' in
          let tm'suc = @t'. NTH_TIME_CHANGES_TRUE (SUC u) (cs_sig_mbm pm') 0 t' in
```

```
        (((ei u, pi u) = (IB_SLAVE_ABS ei' pi' (ti',ti'suc))) /\
         ((pm u, em u) = (MB_MASTER_ABS pm' em' (tm',tm'suc))) /\
         ((er u) = RST_ABS er'))"
   );;

let MT_INTRP_LIVE = new_definition
   ('MT_INTRP_LIVE',
    "! (ei :timeT->pbm_pkt) (em :timeT->mbs_pkt) (pm :timeT->mbm_pkt)
       (pi :timeT->ibs_pkt) (er :timeT->rm_pkt) (ei' :timeC->ib_sin)
       (em' :timeC->mb_min) (er' :timeC->su_mout) (pm' :timeC->mb_mout)
       (pi' :timeC->ib_sout) .
     MT_INTRP_LIVE (ei,em,er) (pm,pi) (ei',em',er') (pm',pi') =
        !ti mti.
        MT_EXEC mti (ei,em,er) (pm,pi) ti
        ==> ?ti'. NTH_TIME_TRUE ti (ale_sig_ibs ei') 0 ti' /\ (ti' > 0)"
   );;

close_theory();;
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Contractor Report

4. TITLE AND SUBTITLE
Interpreter Composition Issues in the Formal Verification of a Processor-Memory Module

5. FUNDING NUMBERS
C NAS1-18586
WU 505-64-50-04

6. AUTHOR(S)
David A. Fura
Gerald C. Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Boeing Company
P.O. Box 3707
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR-4594

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Sally C. Johnson
Final Report

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes interpreter composition techniques suitable for the formal specification and verification of a processor-memory module using the HOL theorem-proving system. The processor-memory module is a multi-chip subsystem within a fault-tolerant embedded system under development within the Boeing Defense and Space Group. Modelling and verification methods were developed that permit provably secure composition at the transaction-level of specification significantly reducing the complexity of the hierarchical verification of the system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Formal Methods; Verification; Microprocessor	72
	16. PRICE CODE: A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		